Learning to Program in Oberon-07

Table of Contents

Learning to Program in Oberon-07 1

Preface 3

Part I — Foundations 7

Chapter 1 — What is a Program? 8

Chapter 2 — Data: Constants and Variables 15

Chapter 3 — Numbers & Arithmetic 21

Chapter 4 — Input and Output 31

Chapter 5 — Making Decisions 41

Chapter 6 — Repetition 55

Chapter 7 — Program Design 67

Part II — Structuring Data 78

Chapter 8 — Arrays 79

Chapter 9 — Characters & Strings 90

Chapter 10 — Records 108

Chapter 11 — Pointers & Dynamic Memory 123

Chapter 12 — Sets 140

Part III — Structuring Programs 151

Chapter 13 — Procedures 152

Chapter 14 — Function Procedures 177

Chapter 15 — Recursion 193

Chapter 16 — Modules 206

Part IV — The OBNC Standard Libraries 217

Chapter 17 — More Input & Output 218

Chapter 18 — The Maths Module 229

Chapter 19 — The Files Module 237

Chapter 20 — Error Handling & Assertions 250

Chapter 21 — Miscellaneous Topics 259

Part V — Putting It All Together 272

Chapter 22 — Object-Oriented Ideas 273

Chapter 23 — A Text-Based Address Book 290

Appendix A — Reserved Words & Symbols 304

Appendix B — Predefined Identifiers & Procedures 307

Appendix C — OBNC Library Reference 310

Appendix D — ASCII Character Table……317

Preface

Programming is the art of instructing a computer to solve problems. Like any art, it requires learning a set of tools and techniques — but more importantly, it requires developing a way of thinking. This book is about both.

The programming language chosen here is Oberon-07, the final language designed by Niklaus Wirth, one of the most influential figures in the history of computing. Wirth spent his career pursuing a single idea: that good programming languages should be small, clean, and free of unnecessary complexity. Oberon-07 is the culmination of that pursuit. It evolved from Pascal and then Modula-2, shedding features along the way rather than accumulating them, until what remained was a language of remarkable elegance. It can be described in full in fewer than twenty pages — yet it is powerful enough to write operating systems, compilers, and real applications.

This makes it an ideal language for learning. There are no dark corners to stumble into, no bewildering array of special cases to memorise. The rules are few and consistent. When a program works, you can understand precisely why. When it does not, the language itself gives you few places to hide.

Einstein's maxim — quoted by Wirth at the head of the Oberon language report — puts it well: *make it as simple as possible, but not simpler.*

Who This Book Is For

This book assumes no prior programming experience. It begins at the very beginning — what a program is, what a computer does with it — and builds from there, one concept at a time. Each new idea is explained before it is used, and every concept is illustrated with complete, runnable programs that you can try for yourself.

If you have programmed before, in any language, you will move through the early chapters quickly. The concepts will be familiar even if the syntax is new. Oberon-07's Pascal heritage means that anyone who has written Pascal or Modula-2 will feel immediately at home.

How This Book Is Organised

The book is divided into five parts.

Part I — Foundations introduces the essential building blocks of every program: data, arithmetic, decision-making, and repetition. By the end of Part I you will be writing programs that interact with the user and solve simple problems.

Part II — Structuring Data introduces the ways Oberon-07 allows you to organise information: arrays, strings, records, pointers, and sets. These are the tools that let you represent real-world things in a program.

Part III — Structuring Programs shows how to break a program into manageable, reusable pieces using procedures, function procedures, recursion, and modules. This is where programs begin to grow beyond a single page.

Part IV — The OBNC Standard Libraries covers the practical tools that come with the OBNC compiler: mathematical functions, file handling, text conversion, error handling, and more. This is

where your programs begin to do genuinely useful things.

Part V — Putting It All Together introduces Oberon-07's approach to object-oriented programming through type extension — a powerful idea that lets you build new types on top of existing ones. The part closes with a complete, substantial project that draws on everything learned in the book.

Each chapter ends with a set of exercises. These are not optional extras — working through them is how the ideas become your own. Solutions, together with the additional modules referenced in the book can be downloaded here:

`https://f005.backblazeb2.com/file/oberon-07/files.zip`

How to Use This Book

Read each chapter in order. The concepts build on one another, and later chapters assume you are comfortable with earlier ones. When you encounter a worked example, do not just read it — type it in, compile it, and run it. Then experiment: change something, see what happens, break it deliberately and observe the error. This active engagement is the fastest route to genuine understanding.

Setting Up OBNC

The programs in this book are written for the OBNC compiler, created by Karl Landström. OBNC is free, open-source, and implements Oberon-07 faithfully. It works by translating your Oberon source code into C, then compiling that C code using your system's C compiler — a process that happens automatically and invisibly when you run the `obnc` command.

OBNC runs on any POSIX-compatible operating system. The following instructions cover the most common setups.

Linux

Linux is the recommended platform for this book, and installation is straightforward. On a Debian-based system (Ubuntu, Linux Mint, and similar), first install the required dependencies:

```
sudo apt install build-essential libgc-dev libsdl1.2-dev
```

Then download the OBNC source package from the official website at `https://miasap.se/obnc/`, extract it, and follow the build instructions in the `README` file. The process amounts to:

```
./build --c-int-type=long
sudo ./install
```

You can verify the installation by typing `obnc` at the terminal — it should print a usage message.

Windows

Windows users are strongly recommended to install **WSL — Windows Subsystem for Linux**. WSL gives you a full Linux environment running inside Windows, and once set up, the Linux instructions

above apply exactly. To install WSL, open PowerShell as Administrator and run:

```
wsl --install
```

This installs Ubuntu by default. Once complete, open the Ubuntu terminal and follow the Linux instructions above. WSL is the most reliable route to a working OBNC installation on Windows.

A pre-compiled Windows-native package of OBNC is also available from the OBNC website, but it has known compatibility issues on some systems. WSL is the more dependable choice.

macOS

macOS is a POSIX-compatible operating system and OBNC can be built from source on it, but this is not officially supported and may require additional steps depending on your system version. If you are comfortable working in a terminal and building software from source, the Linux build instructions are a reasonable starting point. Otherwise, macOS users may find it easier to use a Linux virtual machine.

Your First Program

Once OBNC is installed, open a text editor and type the following:

```
MODULE Hello;
  IMPORT Out;
BEGIN
  Out.String("Hello, World!");
  Out.Ln
END Hello.
```

Save the file as `Hello.obn`. Then, in a terminal in the same directory, compile and run it with:

```
obnc Hello.obn
./Hello
```

If everything is working, you will see:

```
Hello, World!
```

Congratulations — you have just run your first Oberon-07 program. Chapter 1 will explain exactly what every part of it means.

A Note on the Examples

Every code example in this book has been compiled and tested with OBNC on Linux. Most examples are complete, runnable programs. If a program in this book does not compile or behave as described, something has been typed incorrectly; check carefully against the printed version.

One small but important point: Oberon-07 treats uppercase and lowercase letters as distinct. `Begin` is not the same as `BEGIN`, and `integer` is not the same as `INTEGER`. Reserved words in Oberon-07 are always written in full capitals. Pay attention to this from the very first program.

Acknowledgements

This book relies on the work of Niklaus Wirth, who designed Oberon-07 and made the language report freely available. It also relies on Karl Landström's OBNC compiler and its accompanying library documentation, without which a practical introduction to the language would be considerably harder to write. The data abstraction examples in Part V draw directly on Karl Landström's article *Data Abstraction in Oberon*, used with gratitude.

Part I — Foundations

- Chapter 1 — What is a Program?
- Chapter 2 — Data: Constants & Variables
- Chapter 3 — Numbers & Arithmetic
- Chapter 4 — Input and Output
- Chapter 5 — Making Decisions
- Chapter 6 — Repetition
- Chapter 7 — Program Design

Chapter 1 — What is a Program?

Before writing a single line of Oberon-07, it is worth stepping back and asking a more fundamental question: what is a program, and what does a computer actually do with one?

This chapter answers that question. It introduces the ideas of algorithms and instructions, explains what a programming language is and why we need one, and then examines the structure of a complete Oberon-07 program in detail. By the end of the chapter you will understand every part of the "Hello, World!" program shown in the Preface, and you will have written and run a few small variations of your own.

1.1 Algorithms and Instructions

A computer is, at its heart, a machine that follows instructions. It does exactly what it is told — no more, no less. It cannot guess your intentions, fill in gaps, or apply common sense. If your instructions are wrong, the computer will follow them wrongly, faithfully and without complaint.

This sounds like a limitation, but it is actually what makes computers useful. Because a computer follows instructions precisely and tirelessly, it can perform billions of simple operations per second without ever making a mistake of its own. The mistakes, when they occur, belong to the person who wrote the instructions.

The set of instructions that solves a particular problem is called an **algorithm**. The word comes from the name of a ninth-century Persian mathematician, Muhammad ibn Musa al-Khwarizmi, whose works introduced systematic methods for solving mathematical problems to the Western world. An algorithm is not tied to any particular machine or language — it is simply a precise, step-by-step method for accomplishing a task.

Consider a simple everyday example: making a cup of tea. An algorithm for this might be:

1. Fill the kettle with water.
2. Switch the kettle on.
3. Wait until the water boils.
4. Place a teabag in a mug.
5. Pour the boiling water into the mug.
6. Wait two minutes.
7. Remove the teabag.
8. Add milk if desired.

This is an algorithm. It is a finite sequence of unambiguous steps that produces a definite result. Notice that each step is simple and clear. Notice also that the steps must be followed in order — pouring the water before boiling it would not produce good tea.

A computer program is an algorithm expressed in a language that a computer can process. The art of programming is the art of devising correct algorithms and expressing them precisely.

1.2 What a Computer Does

A modern computer consists of several components, but for the purposes of programming, three are most important:

The **processor** (or CPU — Central Processing Unit) is the part that actually executes instructions. It can perform arithmetic, compare values, move data around, and decide which instruction to execute next. It does these things at extraordinary speed — billions of times per second on a modern machine.

Memory (or RAM — Random Access Memory) is where the computer stores the data it is currently working with, as well as the program itself while it is running. Memory is fast but temporary — when you switch the computer off, everything in memory is lost.

Storage (a hard drive or SSD) is where programs and data are kept permanently. When you run a program, it is copied from storage into memory, where the processor can work with it.

When a program runs, the processor fetches each instruction from memory in turn, executes it, and then fetches the next one. This cycle — fetch, execute, fetch, execute — repeats millions of times per second. The entire complexity of modern software emerges from this simple, relentless process.

1.3 Programming Languages

The processor understands only one language: **machine code**, a sequence of numbers that encode instructions directly. Machine code is extraordinarily tedious to write and nearly impossible to read. A single simple task might require dozens of machine code instructions, all expressed as cryptic numeric values.

To solve this problem, programmers invented **high-level programming languages** — languages that are closer to human thought and mathematical notation than to the raw instructions of the processor. A program written in a high-level language must be translated into machine code before the computer can run it. This translation is performed by a program called a **compiler**.

Oberon-07 is a high-level language. When you write an Oberon-07 program and compile it with OBNC, the compiler translates your source code — the text you wrote — into a form the computer can execute directly. You write in a language designed for human clarity; the compiler handles the translation into machine instructions.

Different programming languages make different trade-offs. Some prioritise raw speed. Some prioritise the ability to handle vast complexity. Oberon-07 prioritises clarity and correctness. Its designer, Niklaus Wirth, believed that a program should be easy to read, easy to reason about, and difficult to get wrong. These are the right priorities for learning, and they remain valuable priorities throughout a programming career.

1.4 The Structure of an Oberon-07 Program

Every Oberon-07 program has the same basic structure. Here is a simple complete program:

```
MODULE Hello;
  IMPORT Out;
BEGIN
  Out.String("Hello, World!");
  Out.Ln
END Hello.
```

This program prints the message `Hello, World!` on the screen. Let us examine every part of it.

MODULE and END

An Oberon-07 program is organised as a **module**. A module is a self-contained unit of code with a name. Every module begins with the word `MODULE` followed by the module's name, and ends with the word `END` followed by the same name and a full stop.

```
MODULE Hello;
  ...
END Hello.
```

The name `Hello` is chosen by the programmer. It must match the name of the file the program is saved in — this program must be saved as `Hello.obn`. The `.obn` extension tells OBNC that this file contains Oberon source code.

The word `MODULE` is a **reserved word** — a word that has a fixed, special meaning in Oberon-07 and cannot be used for any other purpose. Reserved words in Oberon-07 are always written entirely in capital letters: `MODULE`, `BEGIN`, `END`, `IMPORT`, and so on. You will encounter all of them as this book progresses.

IMPORT

```
  IMPORT Out;
```

This line tells Oberon-07 that this module uses another module called `Out`. The `Out` module provides procedures for producing output — printing text and numbers on the screen. Without importing it, we would have no way to display anything.

A **module** in Oberon-07 is a collection of related tools — procedures, types, and variables — that can be imported and used by other modules. The `Out` module is part of the standard library that comes with OBNC. We will use it throughout this book.

The semicolon at the end of the line is a separator. In Oberon-07, semicolons separate declarations and statements from one another.

BEGIN and the Statement Sequence

```
BEGIN
  Out.String("Hello, World!");
  Out.Ln
END Hello.
```

The word `BEGIN` marks the start of the module's **statement sequence** — the instructions that will be executed when the program runs. The statements continue until `END Hello.` is reached.

In this program there are two statements.

The first is:

```
Out.String("Hello, World!");
```

This is a **procedure call**. It calls the procedure named `String` from the module `Out`. The dot between `Out` and `String` is how Oberon-07 refers to something that belongs to a particular module: `Out.String` means "the procedure called `String` that belongs to the module called `Out`".

The text in parentheses — `"Hello, World!"` — is the **argument**: the piece of information we are passing to the procedure. Here we are telling `Out.String` what text to print. Text enclosed in double quotation marks is called a **string**.

The second statement is:

```
Out.Ln
```

This calls the procedure `Out.Ln`, which prints a newline — it moves the cursor to the beginning of the next line. Without this, the next thing printed would appear immediately after `Hello, World!` on the same line. It is good practice to end your output with a newline.

Notice that `Out.Ln` has no parentheses and no argument. Some procedures take arguments; others do not. `Out.Ln` needs no information from us — it simply moves to the next line.

Indentation

You may have noticed that the statements inside the module are indented — shifted to the right with spaces. Oberon-07 does not require this; the compiler does not care about indentation. However, indenting your code consistently makes it much easier for a human reader to see the structure at a glance. Throughout this book we will always indent code inside a block of statements, and we encourage you to do the same.

The Full Stop

The final full stop after `END Hello` is not optional. It marks the absolute end of the module. If you omit it, OBNC will report an error.

1.5 Compiling and Running the Program

To compile and run the Hello program, follow these steps.

Open a text editor and type the program exactly as shown. Save it as `Hello.obn`. Open a terminal, navigate to the directory where you saved the file, and type:

```
obnc Hello.obn
```

OBNC will compile the program. If there are no errors, it will create an executable file called `Hello` in the same directory. Run it with:

```
./Hello
```

You should see:

```
Hello, World!
```

If you see an error message instead, read it carefully — OBNC's error messages usually tell you exactly which line contains the problem. The most common mistakes at this stage are: a missing full stop at the end, mismatched capitalisation (remember that `MODULE` and `module` are not the same thing to Oberon-07), or a mismatch between the module name and the filename.

1.6 Comments

A **comment** is a piece of text in the source code that is completely ignored by the compiler. Comments exist for the benefit of human readers — to explain what the code is doing and why.

In Oberon-07, a comment begins with `(*` and ends with `*)`. Everything between these markers is ignored. For example:

```
MODULE Hello;
  IMPORT Out;
BEGIN
  (* Print a greeting to the user *)
  Out.String("Hello, World!");
  Out.Ln  (* Move to a new line *)
END Hello.
```

Comments may appear anywhere in a program. They may span multiple lines:

```
(*
  This is a comment that
  spans several lines.
*)
```

Oberon-07 comments may also be **nested** — a comment may contain another comment inside it. This is unusual among programming languages and is occasionally useful when you want to temporarily disable a block of code that already contains comments.

Get into the habit of writing comments from the very beginning. A program that seemed obvious when you wrote it can become mysterious a few weeks later. A brief comment explaining the purpose of a module or a section of code is always time well spent.

1.7 Why Oberon-07 Is a Good First Language

You might wonder why this book uses Oberon-07 rather than one of the more widely-used languages such as Python, Java, or C. The question is fair.

Popular languages are popular for good reasons — large communities, abundant libraries, clear career paths. But popularity and pedagogical clarity are different things, and a language designed for industrial use is not necessarily the best vehicle for learning to think like a programmer.

Oberon-07 has several qualities that make it particularly well suited for beginners.

It is small. The entire language can be described in fewer than twenty pages. There is no large surface area to memorise before you can do anything meaningful.

It is strict. Oberon-07 insists on clarity and precision. Things that are ambiguous or potentially dangerous in other languages are simply not permitted. This strictness feels restrictive at first, but it catches mistakes early and builds disciplined habits.

It forces you to think. Because Oberon-07 does not try to guess what you meant or silently fix your mistakes, you must understand what you are doing. This is sometimes frustrating, but it produces a deeper understanding than languages that are more forgiving.

The concepts transfer. Everything you learn in Oberon-07 — variables, types, procedures, modules, algorithms — applies directly in any other language you learn afterwards. The ideas are universal; only the syntax differs.

Wirth designed Oberon-07 according to the principle that simplicity is not a compromise — it is a virtue. Learning to program in a clean, simple language first is the surest foundation for everything that follows.

Summary

- A **program** is a precise set of instructions that a computer can execute.
- An **algorithm** is the step-by-step method for solving a problem; a program expresses an algorithm in a language a computer can process.
- A **compiler** translates source code written in a high-level language into a form the computer can execute.
- An Oberon-07 program is organised as a **module**, with a name, an optional import list, and a statement sequence between `BEGIN` and `END`.
- **Reserved words** such as `MODULE`, `BEGIN`, `END`, and `IMPORT` are always written in capital letters.
- **Comments**, enclosed in `(*` and `*)`, are ignored by the compiler and exist for human readers.
- Oberon-07 distinguishes between uppercase and lowercase letters — `MODULE` and `module` are not the same.

Exercises

1.1 Type the Hello program exactly as shown in Section 1.4, compile it with OBNC, and run it. Confirm that you see `Hello, World!` printed on the screen.

1.2 Modify the Hello program to print your own name instead of `Hello, World!`. Compile and run your modified version.

1.3 Add a second `Out.String` call to the program so that it prints two lines of text. What happens if you put `Out.Ln` between the two calls? What happens if you leave it out?

1.4 Deliberately introduce a mistake into the program — for example, change `MODULE` to `Module` or

remove the final full stop — and observe the error message that OBNC produces. Then fix the mistake and confirm the program compiles again. Repeat this with two or three different kinds of mistake so that you become familiar with OBNC's error messages.

1.5 Add a comment to the Hello program explaining what each statement does. Confirm that the program still compiles and runs correctly with the comments present.

1.6 The `Out` module provides a procedure called `Out.Char` that prints a single character. Try adding a call to `Out.Char("!")` at the end of your program, before `Out.Ln`. What do you observe? (Note: a single character in Oberon-07 is enclosed in double quotation marks, just like a string, but contains exactly one character.)

Chapter 2 — Data: Constants and Variables

Every program works with data. A program that calculates your tax bill works with numbers. A program that checks your spelling works with text. A program that plays a game works with the positions of pieces, scores, and the state of the board. Before we can do anything useful in Oberon-07, we need to understand how data is represented and stored.

This chapter introduces the two fundamental ways of holding data in an Oberon-07 program: **constants**, whose values are fixed for the lifetime of the program, and **variables**, whose values can change as the program runs. It also introduces Oberon-07's basic data types — the different kinds of data the language knows about — and the assignment statement, which is the primary tool for giving a variable a value.

2.1 The Concept of a Variable

Imagine a set of labelled boxes. Each box has a name written on the outside and can hold exactly one value at a time. You can look at what is inside a box, replace its contents with something new, or use its contents in a calculation. This is precisely how a variable works in a program.

In more technical terms, a variable is a named location in the computer's memory that holds a value. The name allows us to refer to that location in our code without needing to know anything about where in memory it actually resides — that is the compiler's business, not ours.

Two things are true of every variable:

- It has a **name** (called an identifier) by which we refer to it.
- It has a **type** that determines what kind of value it can hold.

The type is important. A variable that holds whole numbers cannot suddenly be made to hold text. Oberon-07 is strict about this, and for good reason: knowing the type of every variable allows the compiler to catch a whole class of mistakes before the program ever runs.

2.2 Identifiers

An **identifier** is a name chosen by the programmer to refer to something — a variable, a constant, a procedure, a module, or a type. In Oberon-07, identifiers must obey the following rules:

- An identifier consists of letters and digits only.
- It must begin with a letter.
- Uppercase and lowercase letters are considered distinct: `total`, `Total`, and `TOTAL` are three different identifiers.
- An identifier may not be the same as a reserved word. You cannot name a variable `BEGIN` or `END`, for example.

There is no limit on the length of an identifier. The following are all valid identifiers:

```
x
total
numberOfStudents
MaxValue
```

```
item2
```

Choose identifiers that describe what the variable is for. A variable named `total` is far more informative than one named `t`, and a variable named `numberOfStudents` leaves no room for misunderstanding. Clear names cost nothing and save a great deal of confusion.

2.3 Basic Types

Oberon-07 provides several built-in types for representing different kinds of data. The ones we will use most often in the early chapters are:

INTEGER holds whole numbers: values such as 0, 1, -7, 42, and -1000. On a 64-bit system compiled with the `--c-int-type=long` flag (as recommended in the Preface), an INTEGER can hold values from approximately -9.2 × 10^18 to 9.2 × 10^18 — a range more than large enough for any everyday purpose.

REAL holds real numbers — numbers that may have a fractional part, such as 3.14, -0.5, and 2.71828. We will begin using REAL in Chapter 3 once arithmetic has been introduced.

BOOLEAN holds one of exactly two values: `TRUE` or `FALSE`. Boolean variables are used to represent conditions — whether something is the case or not.

CHAR holds a single character: a letter, a digit, a punctuation mark, or any other character in the standard character set. Character literals in Oberon-07 are written in double quotation marks: `"A"`, `"z"`, `"3"`, `"?"`. Note that a single character is written the same way as a short string — the difference is that a CHAR value contains exactly one character.

We will meet further types — BYTE, SET, and others — in later chapters. For now, these four are sufficient.

2.4 Declaring Variables

Before a variable can be used in Oberon-07, it must be **declared**. A declaration introduces the variable's name and specifies its type. Variable declarations appear in a `VAR` section, which comes before the `BEGIN` of a module or procedure.

The syntax of a variable declaration is straightforward:

```
VAR
  name : type;
```

Several variables of the same type can be declared together by separating their names with commas:

```
VAR
  first, second, third : INTEGER;
```

And variables of different types are declared on separate lines (or with separate type specifications):

```
VAR
  count    : INTEGER;
  finished : BOOLEAN;
  initial  : CHAR;
```

2.5 Constants

A **constant** is a name for a fixed value that does not change while the program runs. Constants are declared in a `CONST` section, which appears before the `VAR` section (if there is one). Note carefully that a constant declaration uses a plain equals sign `=`, not the assignment operator `:=` that we will see in Section 2.7:

```
CONST
  MaxScore  = 100;
  PassMark  = 50;
  Separator = "-";
  Passed    = TRUE;
```

The value of a constant must be known at the time the program is compiled — it cannot depend on any calculation that happens at **runtime** (the phase when the executable program is actively executing). Once declared, a constant cannot be changed. Attempting to assign a new value to a constant is an error that the compiler will catch immediately.

Why use constants rather than simply writing the value directly wherever it is needed? There are two good reasons. First, a name like `MaxScore` is more informative than the bare number `100` — it tells the reader what the value represents. Second, if the value ever needs to change, you only need to change it in one place — the constant declaration — rather than hunting through the entire program for every occurrence of `100`.

2.6 The Order of Declarations

In Oberon-07, declarations within a module or procedure must appear in a specific order:

1. `CONST` declarations (if any)
2. `TYPE` declarations (if any — we will meet these in later chapters)
3. `VAR` declarations (if any)
4. Procedure declarations (if any — these come in Part III)

This order is fixed. You cannot put a `VAR` section before a `CONST` section. The compiler will report an error if the order is wrong.

2.7 The Assignment Statement

The **assignment statement** gives a variable a new value. It is written with the assignment operator `:=` (a colon followed by an equals sign), with the variable on the left and the new value on the right:

```
count    := 0;
```

```
finished := FALSE;
grade    := "B";
```

The assignment operator is pronounced *becomes*. So `count := 0` is read as "count becomes zero." This phrasing is deliberately different from the mathematical meaning of equals. In mathematics, `x = 5` asserts that x and 5 are equal. In programming, `x := 5` is an action — it changes the value stored in x to 5.

It is worth pausing on the distinction between `=` and `:=`, since confusing the two is a common mistake. In Oberon-07:

- `=` is used in constant declarations and for comparing two values (as we will see in Chapter 5).
- `:=` is used exclusively for assignment — giving a variable a new value.

The type of the value on the right-hand side of `:=` must be compatible with the type of the variable on the left. You cannot assign a BOOLEAN value to an INTEGER variable, or a CHAR to a BOOLEAN. Oberon-07 is strict about this, and the compiler will refuse to compile a program that violates this rule.

2.8 Example 2.1

Let us bring together what we have learned so far in a complete program. This module uses constants to describe a student's record and variables to hold information that might vary, then prints the results.

```
MODULE StudentRecord;
  IMPORT Out;
CONST
  MaxScore = 100;
  PassMark = 50;
VAR
  studentName : CHAR;
  score       : INTEGER;
  passed      : BOOLEAN;
BEGIN
  studentName := "A";
  score       := 73;
  passed      := TRUE;
  Out.String("Student initial: ");
  Out.Char(studentName);
  Out.Ln;
  Out.String("Score: ");
  Out.Int(score, 0);
  Out.String(" out of ");
  Out.Int(MaxScore, 0);
  Out.Ln;
  Out.String("Pass mark: ");
```

```
  Out.Int(PassMark, 0);
  Out.Ln;
  Out.String("Passed: ");
  Out.Int(ORD(passed), 0);
  Out.Ln
END StudentRecord.
```

Running this program produces:

```
Student initial: A
Score: 73 out of 100
Pass mark: 50
Passed: 1
```

A few things are worth noting. `Out.Int` prints an integer value — it takes two arguments: the value to print, and a minimum field width. Passing `0` for the field width means "use as much space as needed, no more." We will cover all the `Out` procedures fully in Chapter 4.

You will notice that printing the BOOLEAN variable `passed` requires a small trick: `ORD(passed)` converts the BOOLEAN value to an integer — `TRUE` becomes 1 and `FALSE` becomes 0. This is because Oberon-07 does not provide a built-in procedure for printing BOOLEAN values directly. A tidier solution — printing the words "Yes" or "No" — requires the IF statement, which we will learn in Chapter 5.

Also note that `Out.Int(MaxScore, 0)` uses the constant `MaxScore` directly as an argument. Constants can be used anywhere a literal value of the same type could appear.

2.9 Initial Values

When a variable is declared in Oberon-07, its initial value is **undefined**. This means you must not assume that a newly declared variable contains zero, or any other particular value. Before you use a variable in a calculation or print its value, you must assign it a value yourself.

This is another example of Oberon-07's strictness working in your favour. Many bugs in programs written in other languages arise from variables being used before they have been given a meaningful value. By being explicit about this requirement, Oberon-07 encourages the habit of always initialising variables before use — a habit that will serve you well in any language.

Summary

- A **variable** is a named location in memory that holds a value. It has both a name (identifier) and a type.
- An **identifier** consists of letters and digits and must begin with a letter. Oberon-07 is case-sensitive.
- The four basic types introduced in this chapter are: **INTEGER** (whole numbers), **REAL** (numbers with fractional parts — covered fully in Chapter 3), **BOOLEAN** (`TRUE` or `FALSE`), and **CHAR** (a single character).
- Variables are declared in a `VAR` section; constants in a `CONST` section. Declarations must appear

in the order: `CONST`, `TYPE`, `VAR`.

- A constant declaration uses `=`; an assignment statement uses `:=`. These are distinct and cannot be exchanged.
- The **assignment statement** (`:=`) gives a variable a new value. It is an action, not a mathematical equality.
- Variables do not have a defined initial value and must be assigned before use.

Exercises

2.1 Write a program that declares an INTEGER variable called `year`, assigns it the current year, and prints it using `Out.Int`. Compile and run it.

2.2 Declare two INTEGER variables, `a` and `b`. Assign `a` the value 17 and `b` the value 5. Print both values with appropriate labels.

2.3 Write a program that declares a CONST section containing your age as an integer and your initial as a CHAR. Print both values. Confirm that attempting to assign a new value to a constant causes a compiler error.

2.4 Write a program that uses constants to define the number of days in a week (`DaysInWeek = 7`) and the number of weeks in a year (`WeeksInYear = 52`). Declare an INTEGER variable `daysInYear`, assign it the appropriate value, using the constants, and print it (ignore leap years). Hint: the multiplication operator is an asterisk (*).

2.5 Try assigning a BOOLEAN value to an INTEGER variable and observe the error message that OBNC produces. Then try assigning a CHAR value to a BOOLEAN variable. Are the error messages the same or different?

2.6 Declare three INTEGER variables: `x`, `y`, and `temp`. Assign `x` the value 10 and `y` the value 20. Using only assignment statements and the variable `temp` as temporary storage, swap the values of `x` and `y` so that `x` contains 20 and `y` contains 10. Print the values of `x` and `y` before and after the swap using `Out.Int` to confirm it worked.

Chapter 3 — Numbers & Arithmetic

Numbers are at the heart of most programs. Even programs that appear to have nothing to do with mathematics — a word processor, a game, a calendar — are constantly performing arithmetic behind the scenes: counting characters, tracking positions, calculating dates. Before we can write programs of any real interest, we need to understand how Oberon-07 handles numbers and the operations we can perform on them.

This chapter covers two distinct kinds of number — integers and reals — and the arithmetic operations available for each. It explains how Oberon-07 evaluates expressions, how to convert between the two numeric types, and how operator precedence determines the order in which operations are performed. It also introduces two useful predefined functions: `ABS` and `ODD`.

3.1 Expressions and Evaluation

In Chapter 2 we assigned literal values to variables:

```
score := 73;
```

More often we want to compute a value from other values. An **expression** is a combination of values, variables, and operators that the computer evaluates to produce a single result. Here are some examples:

```
2 + 3
score * 2
(a + b) DIV 2
```

When an expression appears on the right-hand side of an assignment, the computer evaluates it first and then stores the result in the variable on the left:

```
average := (a + b) DIV 2;
```

Expressions can be as simple as a single literal value or as complex as you need. The rules for how they are evaluated — which operations happen in which order — are called **operator precedence**, and we will cover them carefully in Section 3.5.

3.2 Integer Arithmetic

The type `INTEGER` represents whole numbers: …, -3, -2, -1, 0, 1, 2, 3, … Oberon-07 provides five arithmetic operators for integers:

Operator	Meaning	Example	Result
`+`	Addition	`7 + 3`	`10`
`-`	Subtraction	`7 - 3`	`4`
`*`	Multiplication	`7 * 3`	`21`

Operator	Meaning	Example	Result
`DIV`	Integer divide	`7 DIV 3`	`2`
`MOD`	Remainder	`7 MOD 3`	`1`

The first three are self-explanatory. The last two deserve some attention.

DIV and MOD

`DIV` performs **integer division**: it divides and discards any fractional part, always rounding towards negative infinity. For positive operands this is the same as ordinary division rounded down:

```
17 DIV 5    (* result: 3, because 17 / 5 = 3.4, rounded down *)
 6 DIV 3    (* result: 2 *)
 1 DIV 4    (* result: 0 *)
```

`MOD` gives the **remainder** after integer division. The result always has the same sign as the divisor (the right-hand operand), and the following relationship always holds:

```
(a DIV b) * b + (a MOD b) = a
```

For positive operands the result is exactly the remainder you would expect from primary school:

```
17 MOD 5  (* result: 2, because 17 = 3 * 5 + 2 *)
 6 MOD 3  (* result: 0, because 3 divides exactly into 6 *)
 1 MOD 4  (* result: 1, because the first operand (1) is less than the second (4) *)
```

`MOD` is particularly useful for tasks such as testing whether a number is even or odd (a number is even if `n MOD 2 = 0`), cycling through a fixed range of values, or converting a number of minutes into hours and minutes.

The divisor in both `DIV` and `MOD` must not be zero. If it is, the program will stop with a runtime error — a **trap**. This is a deliberate design choice: Oberon-07 would rather stop and tell you something has gone wrong than silently produce a meaningless result.

Example 3.1

Here is a program that performs several integer calculations and prints the results:

```
MODULE IntArith;
  IMPORT Out;
VAR
  a, b        : INTEGER;
  sum, diff   : INTEGER;
  product     : INTEGER;
```

```
  quotient    : INTEGER;
  remainder   : INTEGER;
BEGIN
  a := 17;
  b := 5;
  sum       := a + b;
  diff      := a - b;
  product   := a * b;
  quotient  := a DIV b;
  remainder := a MOD b;
  Out.String("a = ");         Out.Int(a, 0);         Out.Ln;
  Out.String("b = ");         Out.Int(b, 0);         Out.Ln;
  Out.String("a + b = ");     Out.Int(sum, 0);       Out.Ln;
  Out.String("a - b = ");     Out.Int(diff, 0);      Out.Ln;
  Out.String("a * b = ");     Out.Int(product, 0);   Out.Ln;
  Out.String("a DIV b = ");   Out.Int(quotient, 0);  Out.Ln;
  Out.String("a MOD b = ");   Out.Int(remainder, 0); Out.Ln
END IntArith.
```

Running this program produces:

```
a = 17
b = 5
a + b = 22
a - b = 12
a * b = 85
a DIV b = 3
a MOD b = 2
```

Note that in this case I have declared variables for the sum, difference, etc to which I assigned the various arithmetic expressions. However, `Out.Int` (and also `Out.Real`, see below) can output the results of arithmetic expressions directly, bypassing the need for a variable. So removing the variables `sum`, `diff`, `product`, `quotient` and `remainder`, and the associated assignments, and replacing the corresponding outputs by:

```
Out.Int(a + b, 0);
Out.Int(a - b, 0);
Out.Int(a * b, 0);
Out.Int(a DIV b, 0);
Out.Int(a MOD b, 0);
```

will give exactly the same results. Of course, in some situations you may need to declare variables because they will be used in further processing elsewhere in the program, or maybe you just want them for clarity. For example, if the arithmetic expression is very complex it's easier to see what its purpose is when the expression is assigned to a variable whose name describes that purpose. For this

reason alone it's generally better to declare variables even for simple expressions, as I've done in the above program.

Unary Minus

The - operator can also appear with a single operand, in which case it negates the value:

```
negA := -a;    (* negA becomes -17 if a is 17 *)
```

This is called the **unary minus**. The + sign can also appear in unary position, though it has no effect.

3.3 Real Arithmetic

The type `REAL` represents numbers with a fractional part: 3.14, -0.5, 2.71828, and so on. Real numbers in Oberon-07 are stored as **floating-point** values — a standard way of representing a wide range of numbers with a fixed number of significant digits. Like all floating-point representations, REAL values are approximations: a number such as one-third cannot be stored exactly. For most practical purposes this does not matter, but it is worth keeping in mind.

Real literals are written with a decimal point or an exponent notation:

```
3.14
-0.5
2.71828
1.0E6       (* 1,000,000 — the E means "times ten to the power" *)
6.022E23    (* Avogadro's number *)
1.5E-3      (* 0.0015 *)
```

Note that a real literal must contain a decimal point or an exponent — the literal `3` is an integer, not a real. To write the real number three, you must write `3.0`.

Oberon-07 provides four arithmetic operators for reals:

Operator	Meaning	Example	Result
`+`	Addition	`3.0 + 1.5`	`4.5`
`-`	Subtraction	`3.0 - 1.5`	`1.5`
`*`	Multiplication	`3.0 * 1.5`	`4.5`
`/`	Division	`7.0 / 2.0`	`3.5`

Note that `/` is used for real division — the result is a real number, and no information is lost. This is in contrast to integer `DIV`, which discards the fractional part. There is no `DIV` or `MOD` for real numbers.

Printing Real Numbers

To print a real value, use `Out.Real`, which takes the value and a minimum field width:

```
Out.Real(3.14159, 0)
```

OBNC's `Out.Real` always produces output in scientific notation — for example, `3.14159` will be printed as something like `3.14159E+00`. This is mathematically correct but may not be the format you expected. This issue will be addressed in a later chapter, but until then, all real output will take this form.

Example 3.2

```
MODULE RealArith;
  IMPORT Out;
VAR
  x, y : REAL;
BEGIN
  x := 7.5;
  y := 2.5;
  Out.String("x = ");       Out.Real(x, 0);       Out.Ln;
  Out.String("y = ");       Out.Real(y, 0);       Out.Ln;
  Out.String("x + y = ");   Out.Real(x + y, 0);   Out.Ln;
  Out.String("x - y = ");   Out.Real(x - y, 0);   Out.Ln;
  Out.String("x * y = ");   Out.Real(x * y, 0);   Out.Ln;
  Out.String("x / y = ");   Out.Real(x / y, 0);   Out.Ln
END RealArith.
```

Running this produces output in the form:

```
x = 7.50000E+00
y = 2.50000E+00
x + y = 1.00000E+01
x - y = 5.00000E+00
x * y = 1.87500E+01
x / y = 3.00000E+00
```

The values are correct — `7.5 + 2.5` is indeed `10`, represented as `1.00000E+01`.

3.4 Mixing Types: FLT and FLOOR

Oberon-07 is strict about types. You cannot use an integer value where a real is expected, or vice versa, without an explicit conversion. This is not an arbitrary restriction — it prevents a whole category of subtle bugs where values silently lose precision or gain it in unexpected ways.

Two predefined procedures handle conversion between `INTEGER` and `REAL`:

`FLT(n)` converts an integer expression `n` to its real equivalent. The name stands for *float*. So if `n` is the integer 3, then `FLT(n)` is the real number `3.0`.

`FLOOR(x)` converts a real expression `x` to an integer by rounding towards negative infinity — that is,

it takes the largest integer not greater than `x`. So `FLOOR(3.7)` is `3`, `FLOOR(-1.2)` is `-2`, and `FLOOR(4.0)` is `4`.

Note the asymmetry: `FLOOR` does not simply truncate towards zero. For negative numbers, it rounds away from zero:

```
FLOOR(2.9)    (* result: 2 *)
FLOOR(2.0)    (* result: 2 *)
FLOOR(-2.1)   (* result: -3, not -2 *)
FLOOR(-2.0)   (* result: -2 *)
```

If you want truncation towards zero for negative numbers, you can use the expression `FLOOR(ABS(x))` with appropriate sign handling, but this is rarely needed in practice; `FLOOR` is the right choice for most purposes. Note: details of `ABS` are given in section 3.6 below.

Why Mixing Types Requires Explicit Conversion

Consider this attempt to compute an average:

```
total := 7;
count := 3;
average := total / count;    (* ERROR: cannot mix INTEGER and REAL with / *)
```

The `/` operator is defined only for real numbers. To compute a real average from integer values, we must convert explicitly:

```
average := FLT(total) / FLT(count);
```

Or, if we are happy with an integer result (the floor of the division), we use `DIV`:

```
intAverage := total DIV count;
```

The explicit conversion makes the programmer's intention clear. There is no silent conversion of integers to reals, no loss of precision that happens without your knowledge. The compiler will refuse to compile code that mixes the types, and you must say precisely what you mean.

Example 3.3

This program converts a temperature from degrees Celsius to degrees Fahrenheit using the formula F = (C × 9/5) + 32:

```
MODULE Temperature;
  IMPORT Out;
VAR
  celsius    : INTEGER;
  fahrenheit : REAL;
```

```
BEGIN
  celsius    := 100;
  fahrenheit := FLT(celsius) * 9.0 / 5.0 + 32.0;
  Out.String("Celsius:    ");
  Out.Int(celsius, 0);
  Out.Ln;
  Out.String("Fahrenheit: ");
  Out.Real(fahrenheit, 0);
  Out.Ln
END Temperature.
```

Running this produces:

```
Celsius:    100
Fahrenheit: 2.12000E+02
```

The value `2.12000E+02` is 212, which is indeed the boiling point of water in Fahrenheit.

3.5 Operator Precedence

When an expression contains more than one operator, Oberon-07 must decide which operation to perform first. These rules are called **operator precedence**.

Oberon-07 divides operators into two precedence levels:

Higher precedence (performed first):

- `*`, `/`, `DIV`, `MOD`

Lower precedence (performed second):

- `+`, `-`

Operators at the same precedence level are evaluated from left to right.

So in the expression `2 + 3 * 4`, the multiplication happens first:

```
2 + 3 * 4    (* = 2 + 12 = 14, NOT 20 *)
```

And in `10 - 3 - 2`, the subtraction proceeds left to right:

```
10 - 3 - 2    (* = 7 - 2 = 5 *)
```

Using Parentheses

To override the default order, use parentheses. Expressions inside parentheses are always evaluated first, regardless of the operators involved:

```
(2 + 3) * 4    (* = 5 * 4 = 20 *)
```

Parentheses can be nested:

```
((2 + 3) * (4 + 1)) DIV 5    (* = (5 * 5) DIV 5 = 25 DIV 5 = 5 *)
```

When in doubt, use parentheses. They make the intended order explicit and leave no room for misreading. Even when they are not strictly required — because the default precedence gives the right answer — they can improve clarity:

```
area := (base * height) DIV 2;    (* clearer than base * height DIV 2 *)
```

Example 3.4

The following program demonstrates how precedence affects results:

```
MODULE Precedence;
  IMPORT Out;
VAR
  a, b, c : INTEGER;
  r1, r2  : INTEGER;
BEGIN
  a  := 10;
  b  := 4;
  c  := 2;
  r1 := a + b * c;         (* = 10 + (4 * 2) = 10 + 8 = 18 *)
  r2 := (a + b) * c;       (* = (10 + 4) * 2 = 14 * 2 = 28 *)
  Out.String("a + b * c   = "); Out.Int(r1, 0); Out.Ln;
  Out.String("(a + b) * c = "); Out.Int(r2, 0); Out.Ln
END Precedence.
```

Running this confirms the results:

```
a + b * c   = 18
(a + b) * c = 28
```

3.7 A Simple Calculator

Let us bring together the ideas of this chapter in a slightly larger program. The following module computes several properties of two numbers and reports them. We will extend this kind of program in Chapter 4 once we have introduced the `In` module for reading values from the user.

```
MODULE Calc;
```

```
  IMPORT Out;
VAR
  a, b             : INTEGER;
  sum, diff        : INTEGER;
  product          : INTEGER;
  quotient         : INTEGER;
  remainder        : INTEGER;
  mean             : REAL;
BEGIN
  a := 25;
  b := 7;
  sum       := a + b;
  diff      := a - b;
  product   := a * b;
  quotient  := a DIV b;
  remainder := a MOD b;
  mean      := FLT(a + b) / 2.0;
  Out.String("a             = "); Out.Int(a, 0);         Out.Ln;
  Out.String("b             = "); Out.Int(b, 0);         Out.Ln;
  Out.String("a + b         = "); Out.Int(sum, 0);       Out.Ln;
  Out.String("a - b         = "); Out.Int(diff, 0);      Out.Ln;
  Out.String("a * b         = "); Out.Int(product, 0);   Out.Ln;
  Out.String("a DIV b       = "); Out.Int(quotient, 0);  Out.Ln;
  Out.String("a MOD b       = "); Out.Int(remainder, 0); Out.Ln;
  Out.String("(a + b)/2.0 = "); Out.Real(mean, 0);       Out.Ln;
END Calc.
```

Running this produces:

```
a             = 25
b             = 7
a + b         = 32
a - b         = 18
a * b         = 175
a DIV b       = 3
a MOD b       = 4
(a + b)/2.0 = 1.60000E+01
```

Note the use of `FLT(a + b)` to convert the integer sum to a real before dividing by `2.0`. Without `FLT`, this would be a type error — you cannot divide an `INTEGER` by a `REAL`. The parentheses around `a + b` ensure that the addition is performed first (with integers), and then the result is converted.

Summary

- An **expression** is a combination of values, variables, and operators that evaluates to a single result.

- Integer arithmetic uses the operators `+`, `-`, `*`, `DIV`, and `MOD`. `DIV` produces the integer quotient (rounded towards negative infinity); `MOD` produces the remainder, with the same sign as the divisor.
- Dividing by zero with `DIV` or `MOD` causes a runtime trap.
- Real arithmetic uses `+`, `-`, `*`, and `/`. Real literals must contain a decimal point or an exponent.
- `Out.Real` in OBNC prints real numbers in scientific notation. A more convenient format will be introduced in Chapter 16.
- `FLT(n)` converts an integer to a real. `FLOOR(x)` converts a real to an integer by rounding towards negative infinity.
- Oberon-07 does not implicitly convert between `INTEGER` and `REAL` — all conversions must be stated explicitly.
- Higher-precedence operators (`*`, `/`, `DIV`, `MOD`) are evaluated before lower-precedence ones (`+`, `-`). Operators at the same level are evaluated left to right. Parentheses override the default order.

Exercises

3.1 Write a program that declares two INTEGER variables, assigns them the values 19 and 4, and prints the results of all five integer operations (`+`, `-`, `*`, `DIV`, `MOD`). Verify the results by hand before running the program.

3.2 Write a program that computes the number of hours and minutes in a given number of minutes. For example, 137 minutes is 2 hours and 17 minutes. Use `DIV` and `MOD`.

3.3 Write a program that converts a distance in kilometres (stored as a REAL) to miles using the conversion factor 1 kilometre = 0.621371 miles. Test it with the value 12.0 kilometres.

3.4 Work out what the following expressions evaluate to, given that `a = 10` and `b = 3`:

```
a + b * 2
(a + b) * 2
a * b MOD 7
(a MOD b) + b
```

Then write a program to check your answers.

3.5 Write a program that uses `FLOOR` to round a REAL value to the nearest integer. The rule for rounding is: add 0.5 to the value, then take `FLOOR`. Test it with the values `2.3`, `2.7`, `-1.3`, and `-1.7`. What results do you expect? Do the results match?

3.6 Write a program that declares an INTEGER variable `n`, assigns it the value 123, and uses `MOD` and `DIV` to extract the individual digits of `n` (the units digit, the tens digit, and the hundreds digit). Print each digit separately.

3.7 Write a program that has two variables: An INTEGER variable ‘numItems’ and a REAL variable ‘costPerItem’. Print out the total cost for 25 items when the costPerItem is £3.49.

Chapter 4 — Input and Output

So far, every program we have written has had its data baked in. The numbers we calculated with, the text we printed — all of it was fixed in the source code and could only be changed by editing the program itself. Real programs are not like that. They receive data from the outside world — from a user typing at a keyboard, from a file, from a network — and they produce results that go somewhere useful.

This chapter is about communication between a program and its user. We will study the `Out` module in full detail, covering all of its procedures and the choices they give you for presenting output. Then we will study the `In` module, which allows a program to read values entered by the user. By the end of the chapter you will be able to write programs that engage in a real, if simple, dialogue with the person running them.

4.1 The Out Module

The `Out` module handles output to the **standard output stream** — in practice, the terminal window where your program runs. We have already used several of its procedures; here we cover all of them systematically.

To use any of these procedures, your module must begin with:

```
IMPORT Out;
```

Out.Ln

```
Out.Ln
```

Writes an end-of-line character and moves the cursor to the beginning of the next line. Takes no arguments. Every sequence of output should end with `Out.Ln` to ensure the terminal prompt appears on a fresh line after the program finishes.

Out.String

```
Out.String(s)
```

Writes the string `s` to the output. The string is written exactly as given, without any surrounding quotation marks. A string literal in Oberon-07 is enclosed in double quotation marks in the source code, but those marks are not part of the string itself — they are only the notation used to write it:

```
Out.String("Hello, World!")    (* prints: Hello, World! *)
Out.String("Score: ")          (* prints: Score:        *)
Out.String("")                 (* prints nothing         *)
```

`Out.String` does not add a newline at the end. If you want the next output to appear on a new line, follow it with `Out.Ln`.

Out.Char

```
Out.Char(ch)
```

Writes a single character `ch` to the output. The argument must be of type `CHAR`:

```
Out.Char("A")      (* prints: A *)
Out.Char("!")      (* prints: ! *)
```

`Out.Char` is useful when you want to print individual characters rather than complete strings — for example, when building output character by character in a loop. We will see it used this way in later chapters.

Out.Int

```
Out.Int(i, n)
```

Writes the integer `i` to the output. The second argument, `n`, specifies a minimum **field width**: if the printed representation of `i` requires fewer than `n` characters, it is padded with spaces on the left. If `n` is zero, the integer is printed using exactly as many characters as it needs, with no padding.

```
Out.Int(42, 0)      (* prints: 42    *)
Out.Int(42, 6)      (* prints:     42  — right-justified in 6 characters *)
Out.Int(-7, 6)      (* prints:     -7  — right-justified in 6 characters *)
Out.Int(12345, 3)   (* prints: 12345  — field width ignored; value needs 5 *)
```

The field width is useful for aligning columns of numbers in a table. A negative sign counts as one of the characters in the field, so `Out.Int(-7, 6)` uses five characters for the number and one for the sign. A plus sign is never written.

For most casual use, pass `0` as the field width and the integer will be printed compactly without padding.

Out.Real

```
Out.Real(x, n)
```

Writes the real number `x` to the output in exponential (scientific) notation. As with `Out.Int`, the second argument `n` specifies a minimum field width, and passing `0` gives a compact result with no padding.

As noted in Chapter 3, OBNC's `Out.Real` always uses exponential notation:

```
Out.Real(3.14159, 0)    (* prints something like: 3.14159E+00 *)
Out.Real(100.0, 0)      (* prints something like: 1.00000E+02 *)
Out.Real(-0.5, 0)       (* prints something like: -5.00000E-01 *)
```

This format is mathematically unambiguous and always shows you the full precision of the stored value. In Chapter 17 we will introduce a module that provides a more convenient procedure for printing real numbers in everyday decimal notation — something like `3.14` rather than `3.14000E+00`.

Out.Hex

```
Out.Hex(i)
```

Writes the integer `i` as an unsigned hexadecimal (base-16) number, with a leading space. This is an OBNC extension not found in all Oberon implementations. We will not use `Out.Hex` in this book; it is mentioned here only for completeness.

A Note on Out.Open

The `Out` module also provides a procedure `Out.Open`, but it does nothing in OBNC — it exists only for compatibility with code written for other Oberon systems. You never need to call it.

Combining Output Calls

Because none of the `Out` procedures add a newline automatically, you compose output by calling them in sequence:

```
Out.String("Result: ");
Out.Int(42, 0);
Out.Ln
```

This produces a single line: `Result: 42`. The semicolons after each statement are separators. The last statement, `Out.Ln`, needs no semicolon because nothing follows it before the `END` of the block — though adding one would not be an error.

This program that demonstrates all the main `Out` procedures together:

```
MODULE OutDemo;
  IMPORT Out;
CONST
  Limit = 100;
VAR
  count : INTEGER;
  rate  : REAL;
  grade : CHAR;
BEGIN
```

```
  count := 42;
  rate  := 98.6;
  grade := "B";
  Out.String("Count:  "); Out.Int(count, 0);    Out.Ln;
  Out.String("Count (width 8): "); Out.Int(count, 8); Out.Ln;
  Out.String("Rate:   "); Out.Real(rate, 0);    Out.Ln;
  Out.String("Grade:  "); Out.Char(grade);       Out.Ln;
  Out.String("Limit:  "); Out.Int(Limit, 0);    Out.Ln
END OutDemo.
```

Running this produces:

```
Count:  42
Count (width 8):       42
Rate:   9.86000E+01
Grade:  B
Limit:  100
```

4.2 The In Module

The `In` module handles input from the **standard input stream** — in practice, text typed by the user at the keyboard. To use it, add `In` to your import list:

```
IMPORT In, Out;
```

Reading input introduces an important new concern: what if the user types something unexpected? What if they type a word when the program expects a number, or press Enter without typing anything? This issue will be dealt with in the next chapter because it requires using the `IF` statement, but for now we'll assume that the user enters valid input so that no error checking is needed.

In.Int

```
In.Int(i)
```

Reads an integer from the input and stores it in the variable `i`. The argument must be a `VAR` parameter — that is, an actual variable, not a literal or expression — because the procedure needs to write into it. (The meaning of `VAR` parameters will be covered fully in Chapter 13; for now, simply note that you pass a variable name directly, without any extra notation.)

`In.Int` skips over any whitespace (spaces, tabs, newlines) before the number. This means that if the user presses Enter a few times, or types spaces before the number, those characters are ignored. The integer may be written in decimal or in hexadecimal with a trailing `H` (for example, `1FH` for 31), though decimal is almost always what you want.

In.Real

```
In.Real(x)
```

Reads a real number from the input and stores it in the variable `x`. Like `In.Int`, it skips preceding whitespace. The number may be written in decimal notation with a point (`3.14`) or in exponential notation (`3.14E+00`). An integer-style literal such as `3` without a decimal point is not accepted as a valid real input.

In.Char

```
In.Char(ch)
```

Reads the next single character from the input — including spaces and newlines — and stores it in `ch`. Unlike the other `In` procedures, `In.Char` does **not** skip whitespace. It reads whatever character is next, whether it is a letter, a space, a digit, or the newline left behind after the user pressed Enter.

This makes `In.Char` useful for reading character-by-character input, but it can also cause surprises: if the user has just pressed Enter to confirm a previous input, the newline character is still sitting in the input stream, and the next call to `In.Char` will read that newline rather than the next thing the user types.

In.String

```
In.String(str)
```

Reads a **quoted string** or a hexadecimal character code from the input and stores it in the character array `str`. A quoted string in Oberon-07 input format is text enclosed in double quotation marks: `"hello"`. The quotation marks are not stored in `str`; only the characters between them are.

In practice, `In.String` is awkward for interactive programs because most users do not think of typing their input in quotation marks. For reading a single word from the user, `In.Name` is usually more convenient.

In.Name

```
In.Name(name)
```

Reads a sequence of non-whitespace characters (a *word* in the everyday sense) and stores it in the character array `name`. It skips leading whitespace, then reads characters until the next whitespace character or end of input. This is typically what you want when asking a user to type a word such as a name, a command, or a short answer.

In.Line

```
In.Line(line)
```

Reads everything from the current position to the end of the current line — including spaces — and stores it in the character array `line`. Unlike the other `In` procedures, `In.Line` does not skip leading whitespace. This is useful when you need to read a complete sentence or a line of mixed content.

In.Open

Like `Out.Open`, `In.Open` exists for compatibility with other Oberon systems. Calling it when the current position is not at the beginning of standard input causes the program to abort, so it is best simply not to call it. OBNC programs do not need it.

4.3 A First Interactive Program

This program reads a number from the user and prints something based on it:

```
MODULE ReadInt;
  IMPORT In, Out;
VAR
  n : INTEGER;
BEGIN
  Out.String("Enter an integer: ");
  Out.Ln;
  In.Int(n);
  Out.String("You entered: ");
  Out.Int(n, 0);
  Out.Ln
END ReadInt.
```

When you run this program and type `42` followed by Enter, it replies:

```
You entered: 42
```

One small observation about prompts: `Out.Ln` is called after the prompt string in this example, which means the cursor moves to the next line and the user types their answer below the prompt. Some programs omit `Out.Ln` so that the user types on the same line as the prompt — like this:

```
Enter an integer: 42
```

Both styles are common. In this book we will generally put the prompt on its own line for clarity, but either is correct.

4.4 Reading Multiple Values

Programs often need to read more than one value. In such a case, each read is a separate call to the appropriate `In` procedure. The user can type each value on its own line, or separate them with spaces on a single line — since `In.Int` and `In.Real` both skip whitespace, both styles work.

```
MODULE ReadTwo;
  IMPORT In, Out;
VAR
  a, b : INTEGER;
BEGIN
  Out.String("Enter two integers: ");
  Out.Ln;
  In.Int(a);
  In.Int(b);
  Out.String("Sum: ");
  Out.Int(a + b, 0);
  Out.Ln
END ReadTwo.
```

If the user types `10 20` (or `10` Enter `20` Enter), the program prints `Sum: 30`.

4.5 Formatting Output with Field Widths

When presenting tabular data, the field width argument of `Out.Int` is invaluable. By using consistent field widths, you can align columns neatly:

```
MODULE Table;
  IMPORT Out;
BEGIN
  Out.String("Number  Square  Cube");
  Out.Ln;
  Out.String("------  ------  ----");
  Out.Ln;
  Out.Int(1, 6);  Out.Int(1,   7);  Out.Int(1,    5);  Out.Ln;
  Out.Int(2, 6);  Out.Int(4,   7);  Out.Int(8,    5);  Out.Ln;
  Out.Int(3, 6);  Out.Int(9,   7);  Out.Int(27,   5);  Out.Ln;
  Out.Int(4, 6);  Out.Int(16,  7);  Out.Int(64,   5);  Out.Ln;
  Out.Int(5, 6);  Out.Int(25,  7);  Out.Int(125,  5);  Out.Ln
END Table.
```

Running this produces a tidy aligned table:

```
Number  Square  Cube
------  ------  ----
  1        1     1
```

```
  2        4     8
  3        9    27
  4       16    64
  5       25   125
```

Each number is right-justified within its column because `Out.Int` pads with spaces on the left. In Chapter 6, once we have introduced loops, we will generate tables like this automatically rather than by repeating statements.

4.6 A Simple Calculator Revisited

We are now in a position to add interactivity to the simple (non-interactive) calculator program presented in the previous chapter. The program asks the user to enter two numbers and then prints the results of the four arithmetic operations on them. We will work with integers and use `FLT` to compute the real quotient.

```
MODULE Calculator;
  IMPORT In, Out;
VAR
  a, b : INTEGER;
BEGIN
  Out.String("Simple Integer Calculator");
  Out.Ln;
  Out.String("--------------------------");
  Out.Ln;
  Out.String("Enter first number:  ");
  Out.Ln;
  In.Int(a);
  Out.String("Enter second number: ");
  Out.Ln;
  In.Int(b);
  Out.String("Results:");
  Out.Ln;
  Out.String("  a + b       = "); Out.Int(a + b, 0);                Out.Ln;
  Out.String("  a - b       = "); Out.Int(a - b, 0);                Out.Ln;
  Out.String("  a * b       = "); Out.Int(a * b, 0);                Out.Ln;
  Out.String("  a DIV b     = "); Out.Int(a DIV b, 0);              Out.Ln;
  Out.String("  a MOD b     = "); Out.Int(a MOD b, 0);              Out.Ln;
  Out.String("  a / b (real)= "); Out.Real(FLT(a) / FLT(b), 0);    Out.Ln
END Calculator.
```

A sample run with inputs `17` and `5`:

```
Simple Integer Calculator
--------------------------
```

```
Enter first number:
17
Enter second number:
5
Results:
  a + b       = 22
  a - b       = 12
  a * b       = 85
  a DIV b     = 3
  a MOD b     = 2
  a / b (real)= 3.40000E+00
```

There is one issue with this program that you may notice: if the user enters `0` for `b`, then `a DIV b` and `a MOD b` will cause an error — division by zero (try it!). Handling this case properly requires the `IF` statement, which we cover in the next chapter. Revisiting and improving this calculator will be one of the exercises there.

4.7 A Note on Interactive Input and Terminals

When you run a program that uses `In`, you are interacting with it through the terminal. The program reads from **standard input**, which is normally connected to your keyboard. The user types their input and presses Enter, and the program reads what was typed.

One subtlety: most terminals operate in **line-buffered** mode, which means the input is not sent to the program until the user presses Enter. So even if your program calls `In.Char` to read a single character, it will wait until the user presses Enter and then read the first character of what was typed. This is the normal behaviour and is not something you need to change.

A second subtlety: after each successful `In.Int` or `In.Real` call, the newline character (from the Enter key) remains in the input stream. If the next read is `In.Char`, it will read that newline rather than whatever the user types next. For this reason, it is safest to stick to `In.Int`, `In.Real`, and `In.Name` in the same program, since all of these skip whitespace (including newlines) automatically. Mixing these with `In.Char` requires care.

Summary

- The `Out` module provides procedures for writing to the terminal. The main ones are: `Out.String` (a string), `Out.Char` (a single character), `Out.Int` (an integer, with field width), `Out.Real` (a real number in scientific notation, with field width), and `Out.Ln` (a newline).
- `Out.Int(i, n)` right-justifies `i` in a field of at least `n` characters, padding with spaces. Passing `0` for `n` prints with no padding.
- The `In` module provides procedures for reading from the keyboard. The main ones are: `In.Int` (an integer), `In.Real` (a real number), `In.Char` (one character, without skipping whitespace), `In.Name` (a whitespace-delimited word), and `In.String` (a quoted string).
- `In.Int`, `In.Real`, and `In.Name` skip preceding whitespace. `In.Char` does not.
- `Out.Real` in OBNC always uses scientific notation. A more convenient formatting procedure will be introduced in Chapter 16.

Exercises

4.1 Write a program that asks the user to enter their age as an integer and then prints `You were born approximately in` followed by the estimated birth year. (Subtract the age from a constant representing the current year.)

4.2 Write a program that reads two real numbers and prints their sum, difference, product, and quotient.

4.3 Write a program that reads two characters using `In.Char` and prints `You entered:` followed by the characters. What is the output when 1) entering the two characters next to each (e.g. AB) and 2) when you enter a space between the characters (e.g. A B)?

4.4 Write a program that reads three integers and prints them in a right-justified column of width 8, one per line, followed by a separator line and their sum:

```
      14
     205
       3
--------
     222
```

(Hint: use `Out.String` for the separator line.)

4.5 Write a program that reads a temperature in Celsius as a real number and prints the equivalent in Fahrenheit, using the formula from Chapter 3. Label both values clearly in the output.

4.6 Write a program that asks the user to enter two integers representing the number of hours and minutes in a time duration, and prints the total number of minutes. For example, if the user enters `2` and `45`, the program should print `165 minutes`.

Chapter 5 — Making Decisions

Every interesting program has to make choices. Should this student pass or fail? Is this number positive, negative, or zero? Which option did the user select? The ability to branch — to follow one path or another depending on the circumstances — is what transforms a fixed sequence of instructions into something that can respond to the world.

This chapter introduces the tools Oberon-07 provides for making decisions: the `BOOLEAN` type and its operators, the `IF` statement, and the `CASE` statement. We will also meet more predefined procedures, including `INC` and `DEC`, which are the idiomatic Oberon way to increment and decrement integer variables.

5.1 Boolean Expressions

A **Boolean expression** is one that evaluates to either `TRUE` or `FALSE`. We met the `BOOLEAN` type briefly in Chapter 2; now we need it in earnest.

Relational Operators

The most common source of Boolean values is a comparison between two values. Oberon-07 provides six **relational operators**, all of which produce a `BOOLEAN` result:

Operator	Meaning	Example	Result
`=`	Equal to	`3 = 3`	`TRUE`
`#`	Not equal to	`3 # 4`	`TRUE`
`<`	Less than	`3 < 4`	`TRUE`
`<=`	Less than or equal to	`4 <= 4`	`TRUE`
`>`	Greater than	`5 > 3`	`TRUE`
`>=`	Greater than or equal to	`3 >= 5`	`FALSE`

These operators work with integers, reals, and characters. For characters, the comparison is based on the ordinal value of the character — its position in the character set — so `"A" < "B"` is `TRUE`, and `"a" > "Z"` is also `TRUE` (because lowercase letters have higher ordinal values than uppercase ones in the standard character set).

Note that the equality operator in Oberon-07 is `=`, not `==` as in some other languages. And the not-equal operator is `#`, not `!=` or `<>`. These are easy to confuse at first; you will quickly get used to them.

Logical Operators

Boolean values can be combined using **logical operators**:

Operator	Meaning	Example	Result
`&`	AND	`TRUE & FALSE`	`FALSE`
`OR`	OR	`TRUE OR FALSE`	`TRUE`
`~`	NOT	`~TRUE`	`FALSE`

`&` (AND) produces `TRUE` only if both operands are `TRUE`. `OR` produces `TRUE` if at least one operand is `TRUE`. `~` (NOT) reverses the value of its single operand.

These operators follow the same precedence rules as arithmetic: `~` (unary) binds most tightly, then `&`, then `OR`. In expressions that mix logical and relational operators, the relational operators are evaluated first (they produce the Boolean values that the logical operators then combine). So the expression:

```
(x > 0) & (x < 10)
```

reads naturally as "x is greater than zero AND x is less than ten." The parentheses are not strictly required by the precedence rules here, but they make the grouping unambiguous and are strongly recommended. When in doubt, parenthesise.

Short-Circuit Evaluation

Oberon-07 evaluates `&` and `OR` with **short-circuit** semantics: the right-hand operand is only evaluated if necessary. For `&`, if the left operand is `FALSE`, the whole expression must be `FALSE` regardless of the right, so the right is not evaluated. For `OR`, if the left operand is `TRUE`, the whole expression must be `TRUE`, so the right is not evaluated.

This matters in practice. If the right-hand side of `&` could cause a runtime error when evaluated (say, dividing by zero), you can prevent this by using an appropriate Boolean expression on the left:

```
(b # 0) & (a MOD b = 0)    (* safe: the MOD is only evaluated if b # 0 *)
```

Boolean Variables

The result of any relational or logical expression can be stored in a `BOOLEAN` variable and used later:

```
VAR
  isAdult  : BOOLEAN;
  age      : INTEGER;
...
age     := 20;
isAdult := age >= 18;    (* isAdult becomes TRUE *)
```

5.2 INC and DEC

Before examining the `IF` statement, it is convenient to introduce two predefined procedures that will appear frequently in examples from here on: `INC` and `DEC`.

`INC(v)` increments the integer variable `v` by one: it is exactly equivalent to `v := v + 1`. **`INC(v, n)`** increments `v` by `n`: equivalent to `v := v + n`.

`DEC(v)` decrements `v` by one: equivalent to `v := v - 1`. **`DEC(v, n)`** decrements `v` by `n`.

```
count := 0;
INC(count);        (* count is now 1 *)
INC(count);        (* count is now 2 *)
INC(count, 10);    (* count is now 12 *)
DEC(count, 3);     (* count is now 9  *)
```

`INC` and `DEC` are the idiomatic Oberon way to modify a counter. Using them instead of writing `v := v + 1` is not merely a style preference — it signals clearly that the purpose of the statement is to step a counter, not to perform a general arithmetic computation.

5.3 The IF Statement

The `IF` statement is the fundamental tool for conditional execution. It evaluates a Boolean expression and, depending on whether it is `TRUE` or `FALSE`, executes one sequence of statements or another. The conditional nature of `IF` means that it acts as a kind of "guard"; if the Boolean expression after `IF` is FALSE, then the code following it is not executed.

The full form of the `IF` statement is:

```
IF expression THEN
  statement sequence
ELSIF expression THEN
  statement sequence
ELSIF expression THEN
  statement sequence
...
ELSE
  statement sequence
END
```

There may be any number of `ELSIF` branches (including none), and the `ELSE` branch is optional. The `END` is always required.

The conditions are evaluated in order, from top to bottom. The first one that evaluates to `TRUE` has its statement sequence executed, and then control passes to whatever follows the `END`. If none of the conditions is `TRUE` and there is an `ELSE` branch, the `ELSE` sequence is executed. If there is no `ELSE` branch and no condition is `TRUE`, nothing happens and execution continues after the `END`.

Simple IF

The simplest form has just a condition and a `THEN` branch:

```
IF score >= 50 THEN
  Out.String("Pass");
  Out.Ln
END
```

Nothing is printed if `score` is less than 50.

IF … ELSE

Adding an `ELSE` handles the case when the condition is false:

```
IF score >= 50 THEN
  Out.String("Pass")
ELSE
  Out.String("Fail")
END;
Out.Ln
```

Note that `Out.Ln` appears after the `END`, outside the `IF`. It will always be executed, regardless of which branch ran. This is the right place to put any statement that should always happen after the decision.

IF … ELSIF … ELSE

Multiple conditions are handled with `ELSIF`:

```
IF score >= 70 THEN
  Out.String("Distinction")
ELSIF score >= 50 THEN
  Out.String("Pass")
ELSE
  Out.String("Fail")
END;
Out.Ln
```

The conditions are tested from the top. If `score` is 85, the first condition is `TRUE` and `Distinction` is printed; the `ELSIF` and `ELSE` branches are skipped. If `score` is 60, the first condition is `FALSE`, the second is `TRUE`, and `Pass` is printed. If `score` is 30, both conditions are `FALSE` and `Fail` is printed.

Nested IF

An `IF` statement can appear inside another `IF` statement. The inner `IF` is indented further to show the structure:

```
IF x > 0 THEN
  IF ODD(x) THEN
    Out.String("positive and odd")
  ELSE
    Out.String("positive and even")
  END
ELSIF x < 0 THEN
  Out.String("negative")
ELSE
  Out.String("zero")
END;
Out.Ln
```

Example 5.1

This program reads a year from the user and determines whether it is a leap year. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years, unless they are also divisible by 400:

```
MODULE LeapYear;
  IMPORT In, Out;
VAR
  year    : INTEGER;
  isLeap  : BOOLEAN;
BEGIN
  Out.String("Enter a year: ");
  Out.Ln;
  In.Int(year);
  IF In.Done THEN
    IF (year MOD 400 = 0) THEN
      isLeap := TRUE
    ELSIF (year MOD 100 = 0) THEN
      isLeap := FALSE
    ELSIF (year MOD 4 = 0) THEN
      isLeap := TRUE
    ELSE
      isLeap := FALSE
    END;
    Out.Int(year, 0);
    IF isLeap THEN
      Out.String(" is a leap year.")
    ELSE
      Out.String(" is not a leap year.")
    END;
    Out.Ln
```

```
  ELSE
    Out.String("Invalid input.");
    Out.Ln
  END
END LeapYear.
```

Testing with 2000 (leap — divisible by 400), 1900 (not leap — divisible by 100 but not 400), 2024 (leap — divisible by 4 but not 100), and 2023 (not leap):

```
2000 is a leap year.
1900 is not a leap year.
2024 is a leap year.
2023 is not a leap year.
```

5.4 A Note on Semicolons

Now that we have statement sequences inside `IF` branches, we must be precise about where semicolons go.

In Oberon-07, a semicolon **separates** statements from one another. It is not a terminator — it does not go at the end of every statement. It goes *between* adjacent statements.

This means:

- There is a semicolon between two consecutive statements.
- There is no semicolon before `ELSIF`, `ELSE`, or `END`, because those are not statements — they are structural keywords that close the current statement sequence.
- There is no semicolon after the last statement in a sequence before a closing keyword.

In practice:

```
IF condition THEN
  statement1;     (* semicolon: separates statement1 from statement2 *)
  statement2      (* no semicolon: END follows, not another statement *)
ELSIF condition2 THEN
  statement3      (* no semicolon: ELSE follows *)
ELSE
  statement4      (* no semicolon: END follows *)
END
```

You may wonder: what happens if you accidentally put a semicolon before `END` or `ELSE`? In Oberon-07, a semicolon followed by `ELSE`, `ELSIF`, or `END` is interpreted as separating the preceding statement from an empty statement — and an empty statement is perfectly valid. So the compiler does not reject it. Both of these are legal:

```
Out.Ln    (* no semicolon before END — correct *)
```

```
END

Out.Ln;   (* semicolon before END — also accepted *)
END
```

This means a stray semicolon before `END` is not a compiler error, but putting one before `ELSE` or `ELSIF` is something to avoid for clarity. The safest habit is: put semicolons between statements, and nowhere else.

5.5 Catching Invalid Input with In.Done

In the previous chapter I mentioned the issue of the user entering unexpected input such as a letter when an integer is required. Now that we have the `IF` statement under our belt we are in a position to use the built-in `BOOLEAN` variable `In.Done`. This can used to catch invalid inputs and provides a way to inform the user of the error. The status of `In.Done` (`TRUE` or `FALSE`) indicates whether the most recent input operation has succeeded. Without it, invalid inputs will result in the program aborting with an error message, or possibly worse: an incorrect value being computed which goes unnoticed. Here is a simple example of its use:

Example 5.2 : Using In.Done

```
MODULE checkInput;
IMPORT In, Out;
VAR
  n : INTEGER;
BEGIN
  Out.String("Enter an integer: ");
  In.Int(n);
  IF In.Done THEN
    Out.String("You entered: ")
    Out.Int(n, 0)
  ELSE
    Out.String("Invalid input!");
  END;
  Out.Ln
END checkInput.
```

We'll see another use for `In.Done` in the next chapter when I introduce looping statements.

5.6 The CASE Statement

The `CASE` statement is an alternative to a long chain of `ELSIF` branches when you need to select between several values of a single expression. Instead of:

```
IF n = 1 THEN
  Out.String("one")
ELSIF n = 2 THEN
  Out.String("two")
ELSIF n = 3 THEN
  Out.String("three")
END
```

you can write:

```
CASE n OF
  1: Out.String("one")
| 2: Out.String("two")
| 3: Out.String("three")
END
```

The vertical bar `|` separates the cases from one another. Each case consists of a label (or list of labels), a colon, and a statement sequence.

What CASE Can Test

The expression after `CASE` must be of type `INTEGER` or `CHAR`. The labels in each branch must be literals or constants of that same type. You cannot use a `BOOLEAN` expression or a variable as a label.

For a `CHAR` expression, the labels are single-character string literals:

```
CASE grade OF
  "A": Out.String("Excellent")
| "B": Out.String("Good")
| "C": Out.String("Satisfactory")
| "D": Out.String("Marginal")
| "F": Out.String("Fail")
END
```

Multiple Labels and Ranges

A single case can match several values by listing them separated by commas:

```
CASE month OF
  1, 3, 5, 7, 8, 10, 12: Out.String("31 days")
| 4, 6, 9, 11:           Out.String("30 days")
| 2:                     Out.String("28 or 29 days")
END
```

A contiguous range of values can be written with `..`:

```
CASE score OF
  90..100: Out.String("A")
| 80..89:  Out.String("B")
| 70..79:  Out.String("C")
| 60..69:  Out.String("D")
END
```

CASE Has No ELSE

Here is the most important thing to know about `CASE` in Oberon-07: **it has no `ELSE` clause.** The language definition does not include one.

This is a deliberate design decision. Wirth's position was that if you write a `CASE` statement, you should know all the possible values the expression can take. If the expression evaluates to a value that is not listed in any branch, the program stops with a runtime trap. The `CASE` statement is not a tool for "handle these values and ignore the rest" — it is a tool for exhaustively selecting among a known set.

This is worth emphasising because other languages (C's `switch`, Pascal's `CASE`) either have an `else`/`default` branch or silently do nothing when no case matches. In Oberon-07, an unlisted value is treated as a programming error.

Working Around the Absence of ELSE

When you genuinely need to handle some values and ignore others, you must guard the `CASE` statement with an `IF`:

```
IF (n >= 1) & (n <= 3) THEN
  CASE n OF
    1: Out.String("one")
  | 2: Out.String("two")
  | 3: Out.String("three")
  END
END
```

The `IF` ensures that the `CASE` is only reached when `n` is one of the three values it handles. Any other value of `n` simply skips the whole construct, and no trap occurs.

This pattern — an `IF` wrapped around a `CASE` — is the standard idiom whenever you want `CASE`-like dispatch without the obligation to cover every value.

Example 5.3 : Converting a test score

Here is a practical example. A test score out of 100 is to be converted to a letter grade. Dividing the score by 10 gives a value in the range 0–10, and each band maps cleanly to a grade — exactly the kind of problem `CASE` with range notation handles well. The `IF` guard rejects scores outside the valid range before the `CASE` is reached:

```
MODULE Grade;
  IMPORT In, Out;
VAR
  score : INTEGER;
  band  : INTEGER;
BEGIN
  Out.String("Enter score (0-100): ");
  Out.Ln;
  In.Int(score);
  IF ~In.Done THEN
    Out.String("Invalid input.");
    Out.Ln
  ELSIF (score < 0) OR (score > 100) THEN
    Out.String("Score must be between 0 and 100.");
    Out.Ln
  ELSE
    band := score DIV 10;
    Out.String("Grade: ");
    CASE band OF
      0..3: Out.String("F  (Fail)")
    | 4:    Out.String("E  (Poor)")
    | 5:    Out.String("D  (Satisfactory)")
    | 6:    Out.String("C  (Good)")
    | 7:    Out.String("B  (Very Good)")
    | 8..9: Out.String("A  (Excellent)")
    | 10:   Out.String("A* (Perfect)")
    END;
    Out.Ln
  END
END Grade.
```

A few things are worth noting. The `..` range notation makes the grade boundaries read almost like a specification: `0..3` covers scores of 0–39, `8..9` covers 80–99. The single value `10` handles the one case where `score DIV 10` reaches 10 — a score of exactly 100. Without that branch, entering 100 would produce a runtime trap, since the `CASE` expression would take the unlisted value 10.

The `IF` guard does two jobs before the `CASE` is reached: it checks that the input was valid, and it checks that the score is in the range 0–100. Once both conditions are satisfied, `score DIV 10` is guaranteed to be in the range 0–10, and the `CASE` covers all eleven possibilities exactly.

When to Use CASE vs IF

Use `CASE` when:

- You are selecting among several specific values of a single integer or character expression.
- You know all the values the expression can legitimately take.
- The alternatives are roughly parallel in structure.

Use IF when:

- The conditions involve ranges, multiple variables, or complex Boolean expressions.
- You need an "otherwise" path and cannot enumerate all cases.
- There are only two or three alternatives.

A CASE statement over three or more specific values is almost always clearer than an equivalent chain of ELSIF. A CASE over two values is usually no clearer than IF ... ELSE, and an IF is simpler to write.

Example 5.4 : A Simple Menu

```
MODULE Menu;
  IMPORT In, Out;
VAR
  choice : INTEGER;
BEGIN
  Out.String("--- Main Menu ---");    Out.Ln;
  Out.String("1. Say hello");         Out.Ln;
  Out.String("2. Say goodbye");       Out.Ln;
  Out.String("3. Print a number");    Out.Ln;
  Out.String("Enter choice (1-3): "); Out.Ln;
  In.Int(choice);
  IF In.Done & (choice >= 1) & (choice <= 3) THEN
    CASE choice OF
      1: Out.String("Hello!")
    | 2: Out.String("Goodbye!")
    | 3: Out.String("The answer is "); Out.Int(42, 0)
    END;
    Out.Ln
  ELSE
    Out.String("Invalid choice.");
    Out.Ln
  END
END Menu.
```

The IF guard serves two purposes: it checks that In.Done is TRUE (the input was valid), and it checks that choice is in the range 1 to 3. Only then does the CASE execute, and since the guard has already established that choice is 1, 2, or 3, the CASE can exhaustively cover all possibilities without risk of a trap.

Example 5.4 : Classifying a Number

Here is a complete program that reads a number, classifies it in several ways, and reports the results. It draws together everything in this chapter: relational and logical operators, IF/ELSIF/ELSE, CASE, and INC.

The program introduces another built-in function: ABS. This function takes an integer or real number

and returns its absolute (positive) value. For example, if the parameter is a negative number such as `-5`, then `ABS(-5)` returns `5`.

The program reports three things: the sign of the number, its parity (odd or even), and a description of its last digit. The last-digit section is a natural fit for `CASE`: `ABS(n) MOD 10` always produces a value in the range 0–9, so the `CASE` can exhaustively list all ten possibilities without needing a guard or risking a trap.

```
MODULE Classify;
  IMPORT In, Out;
VAR
  n        : INTEGER;
  lastDigit: INTEGER;
BEGIN
  Out.String("Enter an integer: ");
  Out.Ln;
  In.Int(n);
  IF ~In.Done THEN
    Out.String("Invalid input.");
    Out.Ln
  ELSE
    (* Sign *)
    Out.String("Sign:       ");
    IF n > 0 THEN
      Out.String("positive")
    ELSIF n < 0 THEN
      Out.String("negative")
    ELSE
      Out.String("zero")
    END;
    Out.Ln;
    (* Parity — only meaningful for non-zero *)
    IF n # 0 THEN
      Out.String("Parity:     ");
      IF ODD(n) THEN
        Out.String("odd")
      ELSE
        Out.String("even")
      END;
      Out.Ln
    END;
    (* Last digit — CASE exhaustively covers 0..9 *)
    lastDigit := ABS(n) MOD 10;
    Out.String("Last digit: ");
    CASE lastDigit OF
      0: Out.String("zero")
    | 1: Out.String("one")
```

```
    | 2: Out.String("two")
    | 3: Out.String("three")
    | 4: Out.String("four")
    | 5: Out.String("five")
    | 6: Out.String("six")
    | 7: Out.String("seven")
    | 8: Out.String("eight")
    | 9: Out.String("nine")
    END;
    Out.Ln
  END
END Classify.
```

A sample run with input `-42`:

```
Sign:       negative
Parity:     even
Last digit: two
```

And with input `305`:

```
Sign:       positive
Parity:     odd
Last digit: five
```

Note that `ABS(n) MOD 10` is always in the range 0–9 regardless of the value of `n`, so no `IF` guard is needed before the `CASE` — unlike the `Menu` example in Section 5.5, where the guard was essential to keep untested values from reaching the `CASE`. Whenever the possible values of the `CASE` expression are bounded by the arithmetic itself, the `CASE` can stand alone safely.

Summary

- **Relational operators** (`=`, `#`, `<`, `<=`, `>`, `>=`) compare two values and produce a `BOOLEAN` result.
- **Logical operators** `&` (AND), `OR`, and `~` (NOT) combine Boolean values. Evaluation is short-circuit: the right operand is only evaluated if necessary.
- `&` has higher precedence than `OR`. Relational operators are evaluated before logical ones. Use parentheses to make grouping explicit.
- **`INC(v)`** increments an integer variable by 1; **`INC(v, n)`** increments by `n`. **`DEC(v)`** and **`DEC(v, n)`** decrement similarly.
- The **`IF`** statement evaluates conditions in order and executes the first branch whose condition is `TRUE`. `ELSIF` and `ELSE` are optional; `END` is always required.
- Semicolons in Oberon-07 **separate** statements. There is no semicolon before `ELSIF`, `ELSE`, or `END`. A semicolon before a closing keyword is accepted (it introduces an empty statement) but is unnecessary.
- The **`CASE`** statement selects among specific values of an integer or `CHAR` expression. Branches

are separated by `|`. Labels may be individual values, comma-separated lists, or ranges written with `..`

- **`CASE` has no `ELSE` clause.** If the expression takes a value not listed in any branch, the program traps at runtime. Guard `CASE` with an `IF` when you need to handle only some values and ignore the rest.

Exercises

5.1 Write a program that reads two integers and prints the larger of the two. If they are equal, print a message saying so.

5.2 Write a program that reads an integer and prints whether it is divisible by 2, by 3, by both, or by neither. (Hint: use `MOD`.)

5.3 Write a program that reads a character using `In.Char` and reports whether it is an uppercase letter (`"A"` to `"Z"`), a lowercase letter (`"a"` to `"z"`), a digit (`"0"` to `"9"`), or something else. Use relational operators, not `CASE`.

5.4 Write a program which reads in two integers and displays a menu :

1. Add
2. Subtract
3. Multiply
4. Divide

The user is then prompted for their choice and the program performs the chosen operation on the numbers.

5.5 Write a program that reads a month number (1–12) and prints the name of the month. Use a `CASE` statement. Guard the `CASE` with an `IF` that checks whether the input is in the range 1 to 12, and print an error message if it is not.

5.6 This exercise introduces the `ODD` function which returns a `BOOLEAN` value. If the parameter `n` is an odd number then `ODD(n)` returns `TRUE`, otherwise it returns `FALSE`. Write a program which reads an integer and tests whether it's even. If it is, the program should determine whether it's also divisible by 4, then print a message informing the user of the results of these tests.

5.7 Write a program that reads an integer and uses `INC` and `DEC` to adjust it: add 1 if the number is negative, subtract 2 if it is positive, and leave it unchanged if it is zero. Print the result.

Chapter 6 — Repetition

Most useful programs do not simply execute a fixed sequence of steps once and stop. They repeat: they process every item in a list, they keep asking until they get a valid answer, they count from one to a hundred. The ability to repeat a sequence of steps — to **loop** — is as fundamental to programming as the ability to make decisions.

Oberon-07 provides three loop constructs, each suited to a different situation:

- `WHILE` — repeat while a condition is true, checking before each iteration.
- `REPEAT` — repeat until a condition is true, checking after each iteration.
- `FOR` — repeat a fixed number of times, stepping through a range of values.

This chapter covers all three in detail, including a distinctive feature of Oberon-07's `WHILE` that sets it apart from the equivalent construct in most other languages.

6.1 The WHILE Statement

The `WHILE` statement repeats a sequence of statements for as long as a condition remains true. The condition is checked **before** each execution of the body:

```
WHILE expression DO
  statement sequence
END
```

If the condition is `FALSE` the very first time it is checked, the body is never executed at all. If the condition is `TRUE`, the body executes, and then the condition is checked again. This continues until the condition is `FALSE`, at which point execution continues after the `END`.

Example 6.1

Here is a simple example of `WHILE` in action — counting from 1 to 5:

```
MODULE Count;
  IMPORT Out;
VAR
  i : INTEGER;
BEGIN
  i := 1;
  WHILE i <= 5 DO
    Out.Int(i, 0);
    Out.Ln;
    INC(i)
  END
END Count.
```

Output:

```
1
2
3
4
5
```

The variable `i` is called the **loop variable** or **counter**. Before the loop, it is set to its starting value. The condition `i <= 5` is the **guard** — the loop continues as long as it is true. Inside the body, `INC(i)` advances the counter. This three-part structure — initialise, guard, advance — is the pattern to look for in every `WHILE` loop.

It is essential that the body of the loop makes progress towards the condition becoming false. If `INC(i)` were omitted from the example above, `i` would never change, the condition would always be true, and the loop would run forever — an **infinite loop**. If this happens to one of your programs, press Ctrl+C in the terminal to interrupt it.

Example 6.2 : Summing a Sequence

Here is a more practical use of `WHILE`: reading integers from the user until they enter a non-number, then printing the sum. `In.Done` value zero acts as a **sentinel** — a special value that signals the end of the input rather than being part of the data. In this case no other sentinel is needed because only numbers are valid inputs, so the loop terminates when a non-number is entered.

```
MODULE Sum;
  IMPORT In, Out;
VAR
  n, total : INTEGER;
BEGIN
  total := 0;
  Out.String("Enter integers, one per line (non-number to finish):");
  Out.Ln;
  In.Int(n);
  WHILE In.Done DO
    INC(total, n);
    In.Int(n)
  END;
  Out.String("Sum: ");
  Out.Int(total, 0);
  Out.Ln
END Sum.
```

Notice the structure: `n` is read once before the loop (the **priming read**), and again at the end of the loop body. The condition checks both that the read succeeded (`In.Done`) and that the value is not the sentinel (`n # 0`). If either is false, the loop ends.

With inputs 4, 7, 2, 0:

```
Sum: 13
```

6.2 The WHILE Statement with ELSIF

Here is a feature of Oberon-07's `WHILE` that surprises programmers coming from other languages: the `WHILE` statement may include `ELSIF` branches, in exactly the same way as an `IF` statement.

The full form is:

```
WHILE expression DO
  statement sequence
ELSIF expression DO
  statement sequence
ELSIF expression DO
  statement sequence
END
```

The semantics, stated precisely in the language report, are: the guards are evaluated in order, and the body of the **first** guard that is true is executed. Then all the guards are evaluated again from the top. The loop continues until **none** of the guards is true.

The LOOP/EXIT Equivalence

To understand why this form exists, it helps to know what it replaces. Some languages provide a `LOOP` statement with an `EXIT` command that breaks out of the loop from the middle. In Oberon-07, there is no `LOOP` statement and no `EXIT`. Instead, the `WHILE`/`ELSIF` form covers the same ground more cleanly.

Consider this `LOOP`/`EXIT` pattern (not valid Oberon-07, shown for illustration):

```
LOOP
  IF b0 THEN
    S0
  ELSIF b1 THEN
    S1
  ELSIF b2 THEN
    S2
  ELSE
    EXIT
  END
END
```

This loop runs `S0` if `b0` is true, `S1` if `b1` is true, `S2` if `b2` is true, and exits when none of them is true. The equivalent in Oberon-07 using `WHILE`/`ELSIF` is:

```
WHILE b0 DO
  S0
ELSIF b1 DO
  S1
ELSIF b2 DO
  S2
END
```

This is not merely a syntactic shorthand — it expresses the intent more directly. The three conditions are the reasons to continue; when none holds, the loop simply ends. There is no need for a separate `EXIT` mechanism.

Example 6.3 : Euclid's Algorithm

A classic algorithm that benefits from the `WHILE`/`ELSIF` form is Euclid's algorithm for computing the greatest common divisor (GCD) of two positive integers. The algorithm works by repeatedly replacing the larger number with the remainder of dividing the larger by the smaller, until both numbers are equal. At that point, the common divisor has been found.

The two-condition version maps naturally onto `WHILE`/`ELSIF`:

```
MODULE GCD;
  IMPORT In, Out;
VAR
  m, n : INTEGER;
BEGIN
  Out.String("Enter two positive integers: ");
  Out.Ln;
  In.Int(m);
  In.Int(n);
  IF In.Done & (m > 0) & (n > 0) THEN
    WHILE m > n DO
      m := m - n
    ELSIF n > m DO
      n := n - m
    END;
    Out.String("GCD = ");
    Out.Int(m, 0);
    Out.Ln
  ELSE
    Out.String("Please enter two positive integers.");
    Out.Ln
  END
END GCD.
```

When `m > n`, the algorithm replaces `m` with `m MOD n`. When `n > m`, it replaces `n` with `n MOD m`. When

neither condition holds, `m = n`, and that value is the GCD. (This is in fact the example given in the Oberon-07 language report itself.)

With inputs 12 and 8:

```
GCD = 4
```

With inputs 100 and 75:

```
GCD = 25
```

The `WHILE/ELSIF` form expresses the algorithm's two symmetric cases cleanly and without any additional structure. Written as a plain `WHILE` with a nested `IF`, the same algorithm would require more careful thought about the loop condition and would be harder to read:

```
WHILE m # n DO
  IF m > n THEN
    m := m - n;
    IF m = 0 THEN m := n END
  ELSE
    n := n - m;
    IF n = 0 THEN n := m END
  END
END
```

The `WHILE/ELSIF` version is not only shorter — it is more directly a transcription of the algorithm's logic.

When to Use WHILE/ELSIF

Use the `WHILE/ELSIF` form when a loop has two or more distinct conditions that each drive a different action, and the loop should continue as long as any one of them is true. It is particularly natural for algorithms that work by alternately reducing two quantities until some balance or termination condition is reached.

For simpler loops with a single condition, the plain `WHILE` is clearer.

6.3 The REPEAT Statement

The `REPEAT` statement is similar to `WHILE`, but it checks its condition **after** each execution of the body:

```
REPEAT
  statement sequence
UNTIL expression
```

The body is always executed at least once. After each execution, the condition is evaluated; if it is `TRUE`, the loop ends. If it is `FALSE`, the body executes again.

REPEAT is the right choice when the body must run at least once before the condition makes sense — most commonly when reading input, since you need to read something before you can check whether it is valid.

Example 6.4 : Validating Input

Here is a program that keeps asking for a number until the user enters something in the valid range:

```
MODULE ReadPositive;
  IMPORT In, Out;
VAR
  n : INTEGER;
BEGIN
  REPEAT
    Out.String("Enter a positive integer: ");
    Out.Ln;
    In.Int(n)
  UNTIL In.Done & (n > 0);
  Out.String("You entered: ");
  Out.Int(n, 0);
  Out.Ln
END ReadPositive.
```

The loop keeps prompting until `In.Done` is true (the input was a valid integer) and `n > 0` (it is positive). Only when both conditions are met does the loop exit.

This is the canonical use of REPEAT: the user must type something before we can check it, so the body must run at least once.

REPEAT vs WHILE

The choice between REPEAT and WHILE comes down to one question: must the body always execute at least once?

- If yes — if the body must run before the condition can even be tested — use REPEAT.
- If no — if the body might legitimately be skipped entirely — use WHILE.

Input validation almost always calls for REPEAT. Processing a list that might be empty calls for WHILE (because if the list is empty, there is nothing to do and the body should not run at all).

Note also that the condition logic is inverted between the two forms: WHILE continues while the condition is TRUE, and REPEAT continues until the condition is TRUE. Equivalently, REPEAT stops when the condition is true; WHILE stops when the condition is false.

Example 6.5 : Digit Reversal

Here is a program that reverses the digits of a positive integer. For example, 1234 becomes 4321. The REPEAT loop is natural here because at least one digit must always be processed:

```
MODULE ReverseDigits;
  IMPORT In, Out;
VAR
  n, reversed, digit : INTEGER;
BEGIN
  Out.String("Enter a positive integer: ");
  Out.Ln;
  In.Int(n);
  IF In.Done & (n > 0) THEN
    reversed := 0;
    REPEAT
      digit    := n MOD 10;
      reversed := reversed * 10 + digit;
      n        := n DIV 10
    UNTIL n = 0;
    Out.String("Reversed: ");
    Out.Int(reversed, 0);
    Out.Ln
  ELSE
    Out.String("Please enter a positive integer.");
    Out.Ln
  END
END ReverseDigits.
```

With input 1234:

```
Reversed: 4321
```

With input 100:

```
Reversed: 1
```

Each iteration peels off the last digit with `MOD 10`, appends it to `reversed` by shifting `reversed` left one decimal place and adding the digit, then removes the last digit from `n` with `DIV 10`. The loop runs until `n` reaches zero — since `n` starts positive and `DIV 10` always reduces it, this is guaranteed to terminate.

6.4 The FOR Statement

The `FOR` statement is the right choice when you know in advance exactly how many times a loop should run, or when you need to step through a specific range of integer values.

```
FOR variable := start TO end DO
  statement sequence
END
```

The **control variable** (here `variable`) takes the value `start` on the first iteration and is incremented by 1 after each iteration. The loop runs while the control variable is less than or equal to `end`. If `start` is greater than `end`, the body is never executed.

The language report defines `FOR v := beg TO end DO S END` as equivalent to:

```
v := beg;
WHILE v <= end DO S; v := v + 1 END
```

This means the `FOR` statement is simply a convenient notation for a common pattern of `WHILE` — it does not add any new capability, but it does make the intent immediately clear.

Example 6.6

```
MODULE Squares;
  IMPORT Out;
VAR
  i : INTEGER;
BEGIN
  Out.String("  i   i*i");
  Out.Ln;
  FOR i := 1 TO 10 DO
    Out.Int(i, 3);
    Out.Int(i * i, 5);
    Out.Ln
  END
END Squares.
```

Output:

```
  i   i*i
  1    1
  2    4
  3    9
  4   16
  5   25
  6   36
  7   49
  8   64
  9   81
 10  100
```

The BY Clause

An optional `BY` clause specifies a step size other than 1. The step must be a **constant expression** (a

literal or a named constant — not a variable):

```
FOR i := 0 TO 20 BY 5 DO
  Out.Int(i, 0); Out.Ln
END
```

Output:

```
0
5
10
15
20
```

The step may be negative, in which case the control variable counts downwards and the loop runs while the variable is greater than or equal to the end value:

```
FOR i := 10 TO 1 BY -1 DO
  Out.Int(i, 0); Out.Ln
END
```

Output:

```
10
9
8
...
1
```

Restrictions on the Control Variable

The `FOR` statement imposes a few important restrictions:

- The control variable must be an `INTEGER` variable declared in the current scope.
- The control variable must not be modified inside the loop body. The loop itself manages its value, and changing it from inside the body would produce undefined behaviour.
- After the loop ends, the value of the control variable is **undefined**. Do not use it after the `END` — its value is not guaranteed to be anything in particular.

These restrictions are not limitations to work around; they are the signal that `FOR` is the right tool specifically for counting loops. If you find yourself wanting to modify the control variable inside the body, or needing its value after the loop, use `WHILE` instead.

Example 6.7 : Computing a Power

Here is a program that computes an integer raised to a non-negative integer power using a `FOR` loop:

```
MODULE Power;
  IMPORT In, Out;
VAR
  base, exponent, result, i : INTEGER;
BEGIN
  Out.String("Base: ");     Out.Ln; In.Int(base);
  Out.String("Exponent: "); Out.Ln; In.Int(exponent);
  IF In.Done & (exponent >= 0) THEN
    result := 1;
    FOR i := 1 TO exponent DO
      result := result * base
    END;
    Out.Int(base, 0);
    Out.String(" ^ ");
    Out.Int(exponent, 0);
    Out.String(" = ");
    Out.Int(result, 0);
    Out.Ln
  ELSE
    Out.String("Please enter a base and a non-negative exponent.");
    Out.Ln
  END
END Power.
```

With inputs 3 and 4:

```
3 ^ 4 = 81
```

With inputs 2 and 10:

```
2 ^ 10 = 1024
```

If the exponent is 0, the loop body never executes (since `1 TO 0` is an empty range), and `result` remains 1 — which is correct, since any number to the power zero is 1. ONBC provides a built-in function to raise a number to a power; we will meet it in chapter 17.

6.5 Choosing the Right Loop

With three loop constructs available, the choice between them is guided by a simple set of questions:

Do you know the number of iterations in advance? If yes, `FOR` is almost certainly the right choice. It states the range and step explicitly and leaves no room for ambiguity.

Must the body execute at least once? If yes, and the number of iterations is not known in advance, use `REPEAT`. Input validation is the archetype.

Might the body need to be skipped entirely? If yes, use `WHILE`. Processing a list that might be

empty, or computing something that might already be done before the loop starts, are typical cases.

Does the loop have two or more distinct continuation conditions, each driving a different action? Use `WHILE/ELSIF`. The GCD algorithm is the model example.

In practice: `FOR` for counting, `REPEAT` for input loops, `WHILE` for everything else.

6.6 Example 6.8 : Summing a Sequence of Numbers

We now have enough tools to write a more complete interactive program. This one reads a sequence of real numbers from the user (terminated by any non-number), computes their count, sum, and average, and prints a summary. Note that, as in example 6.2, only numbers are valid data so it's sufficient for `In.Done` to act as the sentinel; any other input terminates the loop.

```
MODULE Statistics;
  IMPORT In, Out;
VAR
  x, sum        : REAL;
  count         : INTEGER;
BEGIN
  sum   := 0.0;
  count := 0;
  Out.String("Enter real numbers, one per line (a non-number to finish):");
  Out.Ln;
  In.Real(x);
  WHILE In.Done DO
    sum := sum + x;
    INC(count);
    In.Real(x)
  END;
  Out.String("Count: "); Out.Int(count, 0);  Out.Ln;
  Out.String("Sum:   "); Out.Real(sum, 0);   Out.Ln;
  IF count > 0 THEN
    Out.String("Mean:  "); Out.Real(sum / FLT(count), 0); Out.Ln
  ELSE
    Out.String("No data entered.");
    Out.Ln
  END
END Statistics.
```

With inputs 3.5, 7.0, 2.5 :

```
Count: 3
Sum:   1.30000E+01
Mean:  4.33333E+00
```

Summary

- The **WHILE** statement checks its condition before each iteration and runs while the condition is TRUE. The body may never execute if the condition is false from the start.
- Oberon-07's WHILE may include **ELSIF** branches. All guards are evaluated from the top; the body of the first true guard executes, then all guards are re-evaluated. The loop ends when none is true. This replaces the LOOP/EXIT construct found in some other languages.
- The **REPEAT** statement checks its condition after each iteration and runs until the condition is TRUE. The body always executes at least once.
- The **FOR** statement steps an integer control variable through a range from `start` to `end`. An optional BY clause sets the step size (default 1, may be negative). The control variable must not be modified inside the body, and its value is undefined after the loop ends.
- The BY step in a FOR statement must be a constant expression.
- Choose FOR for counting loops, REPEAT for input validation and loops that must run at least once, WHILE for everything else, and WHILE/ELSIF when two or more distinct conditions each drive a different action.

Exercises

6.1 Write a program that uses a FOR loop to print the multiplication table for a number entered by the user — for example, if the user enters 7, print `7 x 1 = 7` through `7 x 12 = 84`, one line per row.

6.2 Write a program that uses a WHILE loop to find the largest power of 2 that is less than or equal to a positive integer entered by the user. For example, for input 100 the answer is 64 (since 2^6 = 64 and 2^7 = 128 > 100).

6.3 Write a program that uses REPEAT to keep asking the user to enter a number between 1 and 10 inclusive, repeating the prompt if the input is out of range or invalid. Once a valid number is entered, print it and stop.

6.4 Write a program that uses a FOR loop with a BY clause to print all even numbers from 2 to 20. Then rewrite it using a plain FOR loop (step 1) and an IF statement inside the body to skip odd numbers. Which version do you prefer, and why?

6.5 Extend the `Statistics` program from Section 6.6 to also print the minimum and maximum values entered. (Hint: initialise `min` and `max` with the first value read, before the loop begins.)

6.6 Write a program that reads a positive integer and prints all of its factors (the integers that divide it exactly). For example, the factors of 12 are 1, 2, 3, 4, 6, and 12. Use a FOR loop from 1 to the number.

6.7 Write a program that uses the WHILE/ELSIF form to simulate a simple traffic light sequence. The light cycles through three states: 0 = red, 1 = green, 2 = amber. Start at red. On each iteration, print the current state and advance to the next (0 ⟶ 1 ⟶ 2 ⟶ 0 …). Use `WHILE state = 0 DO ... ELSIF state = 1 DO ... ELSIF state = 2 DO ...`. Stop after 9 iterations. (Hint: use a separate counter variable to track how many iterations have run, and make all three guards also check that the counter has not reached 9.)

6.8 Write a program that computes the sum of the digits of a positive integer entered by the user. For example, the digit sum of 1234 is 10. Use a REPEAT loop and the MOD/DIV technique from the `ReverseDigits` example.

Chapter 7 — Program Design

Most of the chapters so far have focused on the building blocks of programming. You have learned about variables, expressions, input and output, decisions, and loops. These are the basic tools from which programs are constructed.

However, knowing the tools is not the same as knowing how to solve a problem. A carpenter may own a saw, a hammer, and a chisel, yet still not know how to build a table. Likewise, a programmer may know the syntax of a language and still find it difficult to create useful programs.

The difference is design. Programming is not primarily the act of typing code into a computer; it is not even about syntax: programming is the process of solving problems and expressing those solutions precisely enough for a computer to follow them. The code itself is merely the final product of that process.

Beginners often imagine that experienced programmers somehow know immediately what code to write. In reality, experienced programmers spend most of their time thinking. They analyse the problem, consider alternative solutions, test ideas, and gradually refine a design until the program almost writes itself.

This chapter introduces some of the techniques used to transform a problem into a working program. We will start with a brief review of the concepts covered in previous chapters.

7.1 Review: Building Programs from Simple Parts

Every program you have written so far consists of two fundamental elements: data, stored in variables, and actions, expressed as statements.

Statements manipulate data. For example:

```
total := total + number
```

uses the current values of two variables to calculate a new value. Programs become more interesting when they can make decisions:

```
IF number > 0 THEN
  Out.String("Positive")
END
```

or repeat actions:

```
WHILE number > 0 DO
  Out.Int(number, 0);
  Out.Ln;
  DEC(number)
END
```

Real programs combine these simple ideas in many different ways. A loop may contain decisions. Decisions may contain calculations. Calculations may update variables whose values influence later decisions. The individual parts are simple; the challenge lies in combining them effectively.

7.2 Nested Control Structures

A control structure is a statement that influences the flow of execution. The control structures we have encountered so far are `IF`, `WHILE`, `REPEAT`, and `FOR`. One control structure may appear inside another. This is called **nesting**. We have already seen examples of nesting in the previous chapter; here is another:

Example 7.1

```
MODULE EvenOdd;
  IMPORT Out;
VAR
  i : INTEGER;
BEGIN
  FOR i := 1 TO 10 DO
    IF ODD(i) THEN
      Out.Int(i, 0);
      Out.String(" is odd")
    ELSE
      Out.Int(i, 0);
      Out.String(" is even")
    END;
    Out.Ln
  END
END EvenOdd.
```

The structure can be visualised as:

```
FOR
  IF
  ELSE
  END
END
```

The `IF` statement is nested inside the `FOR` loop. The opposite arrangement is equally common:

```
IF validInput THEN
  WHILE number > 0 DO
    ...
  END
```

```
END
```

Here the loop is nested inside the decision. Programs may contain several levels of nesting:

```
FOR
  IF
    WHILE
    END
  END
END
```

There is no technical limit to how deeply statements may be nested. However, deeply nested programs quickly become difficult to understand. A useful guideline is: keep the structure as simple as possible while still expressing the problem clearly.

In a later chapter we will see how programs may be structured using procedures and function procedures — reusable, named blocks of code designed to perform specific tasks. They are foundational to software development because they break complex programs into smaller, manageable pieces. They also have other important benefits which we will see in Part III.

7.3 Stepwise Refinement

How should we approach a new problem? Many beginners immediately start writing code. Unfortunately, this often leads to confusion because they are trying to solve the problem and write the program at the same time. A more effective approach is to solve the problem first and write the code later.

One of the most important techniques in programming is called **stepwise refinement**. The idea is simple: describe the solution at a very high level, then replace each vague step with more detailed steps, and continue refining until every step can be translated directly into code.

An Everyday Example

We will revisit the making-tea algorithm from Chapter 1. A first description might be:

```
Make tea.
```

This is correct, but far too vague, so we refine it:

```
Boil water.
Put tea in a cup.
Pour the water.
Wait.
```

Now we have more detail, but some steps remain unclear. Refining further:

```
Fill the kettle.
Switch the kettle on.
Put a teabag in the cup.
Wait for the water to boil.
Pour the water.
Wait three minutes.
Remove the teabag.
```

The task has become a sequence of precise actions. Programming works in exactly the same way.

7.4 A First Refinement Example

Suppose we wish to write a program that calculates the sum of several integers. A first description might be:

```
Calculate a sum.
```

This is not yet useful. Refine it:

```
Read numbers.
Add them together.
Display the result.
```

Still too vague. Refine again:

```
Set the total to zero.
Read a number.
While the input is valid:
    Add the number to the total.
    Read another number.
Display the total.
```

With this latest refinement, every action corresponds directly to program statements. The current level of detail is sufficient for translation into Oberon code:

```
total := 0;
In.Int(number);
WHILE In.Done DO
  INC(total, number);
  In.Int(number)
END;
Out.Int(total, 0)
```

Code should always be the final step, not the first step; design comes before implementation.

7.5 Debugging

Even carefully designed programs sometimes produce wrong results or behave unexpectedly. The process of finding and fixing such errors is called **debugging** — a term dating from the early days of computing, when actual insects were occasionally found causing problems in physical hardware.

There are two broad kinds of error. A **syntax error** is a violation of the language's grammar — a missing `END`, a misspelled keyword, an incorrect operator. The compiler detects these and refuses to compile the program until they are fixed. A **logic error** is subtler: the program compiles and runs, but produces the wrong result because the algorithm or its translation into code is incorrect. Logic errors are harder to find because the compiler cannot detect them.

The First Step: Read the Code

Before reaching for any tool, the most effective debugging technique is simply to read the code carefully, as if you were the computer. Many logic errors are visible to a careful reader who is not too close to the code. It helps to read the code as it is written, not as you intended to write it — beginners often read what they meant rather than what they typed.

If reading the code does not reveal the problem, the next step is to trace through it systematically.

Trace Tables

A **trace table** is a systematic record of how a program's variables change as it executes, written out by hand. You step through the program one statement at a time, updating a column for each variable as its value changes. At the end you can see exactly what the program computed and where it diverged from what you expected.

Consider the following program, which is supposed to find the largest value in an array of five integers:

```
MODULE FindMax;
  IMPORT Out;
CONST
  N = 5;
VAR
  data : ARRAY N OF INTEGER;
  max  : INTEGER;
  i    : INTEGER;
BEGIN
  data[0] := 3;
  data[1] := 7;
  data[2] := 2;
  data[3] := 4;
  data[4] := 9;
```

```
  max := data[0];
  FOR i := 1 TO N - 2 DO
    IF data[i] > max THEN
      max := data[i]
    END
  END;
  Out.String("Maximum: ");
  Out.Int(max, 0);
  Out.Ln
END FindMax.
```

The program compiles and runs without error, but prints `Maximum: 7` instead of the correct answer `9`. The bug is not immediately obvious — the initialisation of `max` looks right, and the comparison inside the loop looks right. A trace table reveals the problem.

We draw a column for `i`, `data[i]`, and `max`, and step through the program statement by statement. Each row shows the state of the variables *after* that statement has executed:

Statement	**i**	**data[i]**	**max**
`max := data[0]`	—	—	3
`FOR i := 1 TO 3`, `IF data[1] > max`	1	7	7
`IF data[2] > max`	2	2	7
`IF data[3] > max`	3	4	7
loop ends (`i = 4` never tested)	—	—	7

Notice what the header row of the loop tells us: `N - 2` with `N = 5` gives `3`, so the `FOR` loop only ever takes the values `1, 2, 3`. The largest value, `9`, sits at index `4` — and index `4` is never tested. The trace table makes this visible in a way that reading the code does not: the loop bound `N - 2` *looks* like a reasonable expression, and nothing about the `IF` statement inside the loop is wrong, so the eye tends to slide past the real fault. Writing out each step of execution forces you to confront the fact that `i` stops at `3`, not `4`.

The fix is a single character — changing `N - 2` to `N - 1`, so that the loop visits every remaining index from `1` up to `N - 1` (that is, `4`):

```
FOR i := 1 TO N - 1 DO
```

This is a classic **off-by-one error** — one of the most common bugs in programming, and one that is notoriously hard to spot by eye but immediately obvious in a trace table.

This example illustrates why trace tables are valuable even when the bug is not immediately apparent from reading the code. The act of writing down each step forces you to engage with what the program *actually* does rather than what you *think* it does.

Adding Temporary Output Statements

When a trace table is impractical — because the loop runs hundreds of times, or the values come from user input — you can add temporary `Out` statements to print the values of key variables at critical points in the program. For example, to check that the loop is visiting the correct indices:

```
FOR i := 1 TO N - 2 DO
  Out.String("i = "); Out.Int(i, 0);
  Out.String(", data[i] = "); Out.Int(data[i], 0);
  Out.String(", max = "); Out.Int(max, 0);
  Out.Ln;
  IF data[i] > max THEN
    max := data[i]
  END
END
```

This would immediately show that the loop stops at `i = 3` and never reaches `i = 4`, making the off-by-one error visible without needing to trace by hand. Once the bug is found and fixed, the temporary output statements are removed.

This technique is sometimes called **tracing** or **print debugging**. It is simple but surprisingly effective — experienced programmers use it regularly.

A Debugging Checklist

When a program produces wrong results, work through the following questions:

Is the algorithm correct? Trace through the intended logic on paper, ignoring the code entirely. If the algorithm itself is wrong, fixing the code will not help.

Have all variables been initialised? A variable that is used before it is assigned a value will contain whatever happened to be in memory at that location. Always initialise variables before use.

Are loop bounds correct? Off-by-one errors — loops that run one iteration too many or too few — are among the most common bugs. Check whether the loop should use `N - 1` or `N - 2` as its upper bound, and whether it starts at 0 or 1.

Are conditions correct? Check every `IF` condition carefully. It is easy to write `>` when you meant `>=`, or `&` when you meant `OR`.

Does the output match what you expect at each step? Use a trace table or temporary output statements to check the values of variables at intermediate points, not just at the end.

Debugging is a skill that improves with practice. The most important habit is to approach it methodically — formulating a hypothesis about what is wrong, testing it, and revising the hypothesis based on evidence — rather than making random changes and hoping the problem disappears.

7.6 Worked Example — Designing a Number Guessing Game

Let us apply stepwise refinement to a larger problem, one in which the benefits of the technique are more clearly evident. We wish to create a game in which the computer chooses a random number,

the user repeatedly guesses it, the computer indicates whether each guess is too high or too low, and the game ends when the correct number is guessed.

First Refinement

```
Play a guessing game.
```

Second Refinement

```
Choose a secret number.
Ask for guesses.
Report success.
```

Third Refinement

```
Choose a secret number.

Repeat:
    Ask for a guess.
    Compare the guess with the secret number.
Until the guess is correct.
```

Fourth Refinement

```
Choose a secret number.

Repeat:
    Ask for a guess.
    If the guess is too low:
        Report this.
    Otherwise if the guess is too high:
        Report this.
    Otherwise:
        Report success.
Until the guess is correct.
```

At this stage the design is almost complete. One practical issue remains: the random number. OBNC does not include a random number generator in its standard library, but I have written a small `Rand` module that provides one. See the preface of this book for the download link. Place the module in the same directory as the program below and compile as normal.

The Program

```
MODULE Guess;
  IMPORT In, Out, Rand;
VAR
  secret, guess, attempts : INTEGER;
BEGIN
  secret   := Rand.Range(100) + 1;
  attempts := 0;
  Out.String("I am thinking of a number between 1 and 100.");
  Out.Ln;
  REPEAT
    Out.String("Your guess: ");
    Out.Ln;
    In.Int(guess);
    INC(attempts);
    IF guess < secret THEN
      Out.String("Too low.")
    ELSIF guess > secret THEN
      Out.String("Too high.")
    ELSE
      Out.String("Correct!")
    END;
    Out.Ln
  UNTIL guess = secret;
  Out.String("You needed ");
  Out.Int(attempts, 0);
  Out.String(" guesses.");
  Out.Ln
END Guess.
```

A sample session might look like:

```
I am thinking of a number between 1 and 100.
Your guess:
50
Too low.
Your guess:
75
Too high.
Your guess:
63
Too low.
Your guess:
68
Correct!
```

```
You needed 4 guesses.
```

The final program appears substantial, but every part emerged naturally from a series of refinements.

7.7 A Programmer's Checklist

When faced with a new programming problem, the following checklist is often helpful.

Understand the problem. Before writing anything, ask: what information is available? What result is required? What are the inputs and outputs? A clear understanding of the goal is more valuable than any amount of code.

Identify the data. What values must be stored? What variables are needed, and of what types?

Describe the algorithm. Write the solution in plain language. Avoid thinking about syntax at this stage. The goal is to express what the program must do, not how to write it in Oberon.

Refine the design. Apply stepwise refinement: replace large steps with smaller steps, and continue until every step is precise enough to be translated directly into code.

Translate into code. Only now should you begin writing Oberon. If the refinement has been thorough, this step is largely mechanical.

Test carefully. Try typical values, boundary values (the smallest and largest inputs the program should handle), and invalid inputs. Many bugs reveal themselves only at the edges.

Improve. The first version of a program is rarely the final version. Once it works, look for opportunities to simplify, clarify, and extend the design.

Summary

Programming is the process of turning a problem into a sequence of precise instructions. With very small programs you can often get away without much thought given to the design, but with programs longer than a few lines, design is essential. Begin with understanding, identify the necessary data, develop an algorithm, refine the design, and only then translate the result into code.

The most important technique introduced in this chapter is **stepwise refinement**. Large problems become manageable when they are broken into smaller problems and solved one step at a time. This technique will remain useful throughout the rest of this book and throughout your programming career.

Exercises

7.1 Use stepwise refinement to design a program that reads a sequence of positive integers (terminated by 0) and prints the largest value entered. Write out at least three levels of refinement before writing any code, then implement and test the program.

7.2 Use stepwise refinement to design a program that reads an integer and prints all of its factors (the integers that divide it exactly). For example, the factors of 12 are 1, 2, 3, 4, 6, and 12. Write out the refinement steps before coding.

7.3 A program is needed to convert a time given as a total number of seconds into hours, minutes, and seconds. For example, 3661 seconds is 1 hour, 1 minute, and 1 second. Apply the checklist from Section 7.6: identify the inputs and outputs, name the variables needed, write the algorithm in plain language, then translate it into Oberon.

7.4 Use stepwise refinement to design a program that reads 10 integers and reports the count of positive values, the count of negative values, and the count of zeros. Write at least two levels of refinement, then implement the program.

7.5 The number-guessing game in Section 7.6 does not handle invalid input — if the user types a word instead of a number, the program will behave unexpectedly. Using what you know about `In.Done`, extend the program so that it detects invalid input and asks the user to try again, without counting the invalid attempt.

Part II — Structuring Data

- Chapter 8 — Arrays
- Chapter 9 — Characters & Strings
- Chapter 10 — Records
- Chapter 11 — Pointers & Dynamic Memory
- Chapter 12 — Sets

Chapter 8 — Arrays

So far, every variable we have used has held exactly one value at a time. A variable of type `INTEGER` holds one integer; a variable of type `REAL` holds one real number. This is fine for simple calculations, but many real problems involve collections of values: a list of examination scores, a set of temperature readings, the pixels of an image. Handling these with individual variables would be impractical. Imagine declaring a hundred separate variables just to store a hundred numbers — and then writing separate statements to process each one.

The solution is the **array**: a single variable that holds multiple values of the same type, arranged in a sequence and accessed by position. Arrays are one of the most fundamental data structures in programming, and understanding them well is essential preparation for everything that follows in Part II.

8.1 What Is an Array?

An array is a fixed-size, ordered collection of values, all of the same type. Think of it as a row of numbered boxes, each capable of holding one value. The boxes are numbered starting from zero, and you access any individual box by specifying its number — its **index**.

For example, an array of five integers might look like this:

```
Index:   0    1    2    3    4
Value:  14   27    3   91   56
```

To get the value at position 2, you write the array name followed by the index in square brackets: `a[2]`, which gives `3`. To change the value at position 4, you write `a[4] := 99`.

Two things are fixed for the lifetime of an array: its **size** (the number of elements it contains) and the **type** of its elements. You cannot add more elements to an array once it has been declared, and you cannot store a value of the wrong type in it.

8.2 Declaring Arrays

An array variable is declared in the `VAR` section using the following syntax:

```
VAR
  name : ARRAY size OF type;
```

where `size` is the number of elements (a positive integer constant) and `type` is the type of each element. For example:

```
VAR
  scores    : ARRAY 5 OF INTEGER;
  readings  : ARRAY 10 OF REAL;
  flags     : ARRAY 8 OF BOOLEAN;
```

`scores` is an array of 5 integers. `readings` is an array of 10 reals. `flags` is an array of 8 Boolean values.

The size must be a constant — you cannot use a variable as the size. This is because the compiler needs to know the size at **compile time** (the phase when your source code is translated into machine code) in order to allocate the correct amount of memory. If you need a size that is determined at runtime, that requires a different approach (dynamic memory, covered in Chapter 11).

It is good practice to define the size as a named constant rather than a bare number. This way, if you ever need to change the size, you only need to change it in one place:

```
CONST
  N = 5;
VAR
  scores : ARRAY N OF INTEGER;
```

8.3 Indexing: Accessing Individual Elements

Individual elements of an array are accessed using square brackets containing the index:

```
scores[0]    (* the first element *)
scores[1]    (* the second element *)
scores[4]    (* the fifth (last) element of a 5-element array *)
```

Oberon-07 arrays are zero-indexed. The first element is always at index `0`, and the last element of an array of size `n` is at index `n - 1`. This is the same convention used in C, Java, and most modern languages, though it differs from Pascal (where arrays can start at any index you choose).

An array element can be used anywhere an ordinary variable of the same type can be used — on the left-hand side of an assignment, in an expression, or as an argument to a procedure:

```
scores[0] := 74;
scores[1] := scores[0] + 10;
Out.Int(scores[1], 0);
```

Out-of-Bounds Access

What happens if you use an index that is outside the valid range — say, `scores[5]` for a 5-element array, or `scores[-1]`? In Oberon-07, this is a runtime trap: the program stops immediately with an error. This is the correct behaviour — accessing memory outside an array's bounds is a serious error that can cause unpredictable results in languages that do not check for it. Oberon-07's strictness here protects you from a whole class of subtle bugs.

Using a Variable as an Index

The index in square brackets does not have to be a literal — it can be any integer expression, including a variable:

```
i := 2;
scores[i] := 88;        (* same as scores[2] := 88 *)
scores[i + 1] := 91;    (* same as scores[3] := 91 *)
```

This is what makes arrays genuinely powerful: by varying the index, a single statement can operate on any element of the array. Combined with a loop, this lets you process every element with very little code.

8.4 Arrays and FOR Loops

The natural companion to an array is the `FOR` loop. Because array indices are integers and `FOR` steps an integer variable through a range, the two fit together perfectly.

Example 8.1: Initialising an Array

This program fills an array with the squares of the integers from 1 to 5:

```
MODULE Squares;
  IMPORT Out;
CONST
  N = 5;
VAR
  sq : ARRAY N OF INTEGER;
  i  : INTEGER;
BEGIN
  FOR i := 0 TO N - 1 DO
    sq[i] := (i + 1) * (i + 1)
  END;
  FOR i := 0 TO N - 1 DO
    Out.Int(i + 1, 0);
    Out.String(" squared = ");
    Out.Int(sq[i], 0);
    Out.Ln
  END
END Squares.
```

Output:

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

The first loop fills the array; the second prints it. The index variable `i` runs from `0` to `N - 1` — the

valid range for an array of size `N`. Note the pattern `FOR i := 0 TO N - 1 DO`: this is the standard idiom for iterating over all elements of an array of size `N`, and you will see it constantly.

Example 8.2: Finding the Maximum

A classic array task is finding the largest value in a collection. The standard approach is to initialise a `max` variable with the first element, then scan the remaining elements, updating `max` whenever a larger value is found:

```
MODULE Maximum;
  IMPORT Out;
CONST
  N = 6;
VAR
  data : ARRAY N OF INTEGER;
  max  : INTEGER;
  i    : INTEGER;
BEGIN
  data[0] := 34;
  data[1] := 12;
  data[2] := 78;
  data[3] := 5;
  data[4] := 56;
  data[5] := 23;
  max := data[0];
  FOR i := 1 TO N - 1 DO
    IF data[i] > max THEN
      max := data[i]
    END
  END;
  Out.String("Maximum: ");
  Out.Int(max, 0);
  Out.Ln
END Maximum.
```

Output:

```
Maximum: 78
```

The loop starts at index 1 because the first element has already been used to initialise `max`. There is no point comparing `data[0]` with itself.

8.5 The LEN Function

The predefined function `LEN` returns the number of elements in an array:

```
LEN(a)
```

For an array `a` declared as `ARRAY N OF INTEGER`, `LEN(a)` returns `N`. This allows you to write loops that work correctly regardless of the array's size, without needing to refer to the constant `N` directly:

```
FOR i := 0 TO LEN(scores) - 1 DO
  (* process scores[i] *)
END
```

Using `LEN` is particularly valuable when you pass arrays to procedures (covered in Chapter 13), where the size may not be directly visible. For now, either `N - 1` or `LEN(a) - 1` as the upper bound of a `FOR` loop is fine — they give the same result. The habit of using `LEN` is worth developing.

8.6 Reading an Array from the User

Arrays become much more useful when their contents come from the user rather than being fixed in the source code. Here is a program that reads `N` integers and stores them in an array:

```
MODULE ReadArray;
  IMPORT In, Out;
CONST
  N = 5;
VAR
  data : ARRAY N OF INTEGER;
  i    : INTEGER;
BEGIN
  Out.String("Enter ");
  Out.Int(N, 0);
  Out.String(" integers:");
  Out.Ln;
  FOR i := 0 TO N - 1 DO
    In.Int(data[i])
  END;
  Out.String("You entered:");
  Out.Ln;
  FOR i := 0 TO N - 1 DO
    Out.String("  data[");
    Out.Int(i, 0);
    Out.String("] = ");
    Out.Int(data[i], 0);
    Out.Ln
  END
END ReadArray.
```

With inputs 10, 30, 20, 50, 40:

```
Enter 5 integers:
10
30
20
50
40
You entered:
  data[0] = 10
  data[1] = 30
  data[2] = 20
  data[3] = 50
  data[4] = 40
```

Notice how `In.Int(data[i])` uses the array element directly as the argument — since `In.Int` requires a `VAR` parameter (a variable it can write into), and an indexed array element is exactly that.

8.7 Multi-Dimensional Arrays

An array's elements can themselves be arrays. The result is a **multi-dimensional array** — most commonly a two-dimensional array, which can be thought of as a table with rows and columns.

The declaration looks like this:

```
VAR
  grid : ARRAY 3 OF ARRAY 4 OF INTEGER;
```

This declares a 3×4 array — 3 rows and 4 columns, for a total of 12 integer values. Alternatively, Oberon-07 allows a more compact notation:

```
VAR
  grid : ARRAY 3, 4 OF INTEGER;
```

Both forms are equivalent. An element is accessed with two indices:

```
grid[1][2] := 42;    (* row 1, column 2 *)
```

Or, with the compact declaration, the same element can be written as:

```
grid[1, 2] := 42;
```

Example 8.3: A Multiplication Table

This program fills a 4×4 multiplication table and prints it:

```
MODULE MultTable;
  IMPORT Out;
CONST
  N = 4;
VAR
  table : ARRAY N, N OF INTEGER;
  i, j  : INTEGER;
BEGIN
  (* Fill the table *)
  FOR i := 0 TO N - 1 DO
    FOR j := 0 TO N - 1 DO
      table[i, j] := (i + 1) * (j + 1)
    END
  END;
  (* Print the table *)
  FOR i := 0 TO N - 1 DO
    FOR j := 0 TO N - 1 DO
      Out.Int(table[i, j], 5)
    END;
    Out.Ln
  END
END MultTable.
```

Output:

```
    1    2    3    4
    2    4    6    8
    3    6    9   12
    4    8   12   16
```

The nested `FOR` loops are the standard idiom for processing a two-dimensional array: the outer loop steps through rows, the inner loop steps through columns. The field width of `5` in `Out.Int` keeps the columns aligned.

Note that each element `table[i, j]` is filled before being printed. In this particular program the table could be printed without storing the values at all (just computing `(i+1)*(j+1)` directly in the print loop), but storing the table first is the more general pattern — it allows the data to be used in multiple ways after it is computed.

8.8 Example: Averaging a List of Numbers

We now have enough tools to write a program that reads a list of numbers from the user, stores them in an array, and then computes and reports the count, sum, minimum, maximum, and average.

The key advantage of storing the numbers in an array (rather than processing them in a single pass as in Chapter 6) is that the data is available for multiple traversals. Here we only need two passes — one to read the data, one to compute the statistics — but in later chapters we will see tasks that

require the stored data directly (such as sorting, covered in Chapter 13's exercises).

```
MODULE Stats;
  IMPORT In, Out;
CONST
  MaxN = 20;
VAR
  data : ARRAY MaxN OF REAL;
  n : INTEGER;
  i : INTEGER;
  sum, min, max : REAL;
  x : REAL;
BEGIN
  (* Read the data *)
  n := 0;
  Out.String("Enter up to ");
  Out.Int(MaxN, 0);
  Out.String(" real numbers (a non-number to finish):");
  Out.Ln;
  In.Real(x);
  WHILE In.Done & (n < MaxN) DO
    data[n] := x;
    INC(n);
    In.Real(x)
  END;
  IF n = 0 THEN
    Out.String("No data entered.");
    Out.Ln
  ELSE
    (* Compute statistics *)
    sum := data[0];
    min := data[0];
    max := data[0];
    FOR i := 1 TO n - 1 DO
      sum := sum + data[i];
      IF data[i] < min THEN min := data[i] END;
      IF data[i] > max THEN max := data[i] END
    END;
    Out.String("Count:   "); Out.Int(n, 0);              Out.Ln;
    Out.String("Sum:     "); Out.Real(sum, 0);            Out.Ln;
    Out.String("Minimum: "); Out.Real(min, 0);            Out.Ln;
    Out.String("Maximum: "); Out.Real(max, 0);            Out.Ln;
    Out.String("Mean:    "); Out.Real(sum / FLT(n), 0);  Out.Ln
  END
END Stats.
```

With inputs 3.0, 7.5, 2.0, 9.5, 4.0:

```
Count:   5
Sum:     2.60000E+01
Minimum: 2.00000E+00
Maximum: 9.50000E+00
Mean:    5.20000E+00
```

A few points are worth noting. The array is declared with size `MaxN` — the maximum number of values we are willing to accept — but the actual number entered by the user may be less than this. The variable `n` tracks how many elements have actually been stored; subsequent processing uses `n` as the upper bound, not `MaxN`. This pattern — an array with a declared maximum size and a separate count variable — is extremely common in programs that read a variable-length list into a fixed-size array.

The loop condition `In.Done & (n < MaxN)` has two parts: it stops if the input is not a valid real number (the sentinel), and it stops if the array is full. Both checks are necessary to prevent an out-of-bounds access.

The statistics are initialised from `data[0]` before the loop, which then starts at index `1`. This is the same pattern used in the `Maximum` example in Section 8.4 — correct initialisation from the first element avoids the need for special cases.

8.9 Copying and Assigning Arrays

In Oberon-07, you can assign one array to another with a single assignment statement, provided both arrays have the same type and size:

```
VAR
  a, b : ARRAY 5 OF INTEGER;
...
a := b;    (* copies all 5 elements of b into a *)
```

After this assignment, `a` and `b` hold the same values, but they are independent copies — modifying an element of `a` afterwards does not affect `b`. This is different from the behaviour of pointers, which we will encounter in Chapter 11.

You cannot assign arrays of different sizes, even if the element type is the same. The compiler will refuse it.

Summary

- An **array** is a fixed-size, ordered collection of values all of the same type. It is declared as `ARRAY size OF type`.
- Array elements are accessed by **index**, starting at `0`. The last element of an array of size `n` is at index `n - 1`.
- Accessing an element with an out-of-bounds index causes a **runtime trap**.
- The predefined function **`LEN(a)`** returns the number of elements in array `a`.

- The standard idiom for processing every element of an array is `FOR i := 0 TO LEN(a) - 1 DO ... END`.
- A **multi-dimensional array** is declared as `ARRAY m, n OF type` (or equivalently as `ARRAY m OF ARRAY n OF type`). Elements are accessed as `a[i, j]`.
- Nested `FOR` loops are the standard way to process a two-dimensional array.
- For variable-length lists stored in a fixed-size array, use a separate integer variable to track how many elements have actually been stored.
- Arrays of the same type and size can be copied with a single assignment statement.

Exercises

8.1 Write a program that declares an array of 10 integers, fills it with the values 1 to 10 using a `FOR` loop, and then prints them in reverse order (10 down to 1).

8.2 Write a program that reads in 10 integers into an array and counts how many of them are positive, how many are negative, and how many are zero. Print the three counts.

8.3 Write a program that declares two arrays of 5 integers each, fills the first with values entered by the user, copies it to the second using a `FOR` loop (element by element, not with a single array assignment), and then prints both arrays side by side using field widths to align the columns.

8.4 Write a program that reads up to 15 real numbers into an array (stopping when a non-number is entered) and then computes and prints the sum of only the positive values.

8.5 In this exercise you are asked to compute the **Standard Deviation** of a set of 10 numbers input by the user. The standard deviation is a measure of the "spread" of a set of numbers and is calculated with the following algorithm:

1. Calculate the average of the set of numbers.
2. Now look at each of the numbers in your set one by one. Subtract the average computed in step 1 from each number to find out how far away it is from the center. (Note: some of these numbers will be negative, and that's fine).
3. Take each of the distances found in step 2 and multiply it by itself (square it). This will turn all negative numbers found in step 2 to positive numbers.
4. Add all the squared numbers together into one big total.
5. Divide the total calculated in step 4 by the number of numbers in the set, minus one. e.g. if there were 10 numbers in the set you should divide the total by 9.
6. Take the square root of the number found in step 5. This is the standard deviation.

Write a program which implements the standard deviation algorithm. You will need to import the `Math` library in order to calculate the square root. Just add `Math` to the `IMPORT` list and use `Math.sqrt(X)` to compute the square root, where `X` is the number found in step 5. Test your program with the following set of numbers : 1,2,3,4,5,6,7,8,9,10. The program should output 3.0277 (rounded).

8.6 Write a program that declares a 3×3 two-dimensional array of integers, fills it so that `table[i, j] = i * 3 + j + 1` (giving the values 1 through 9), and prints it as a 3×3 grid with each number right-justified in a field of width 4.

8.7 Write a program that reads a 3×3 grid of integers from the user (row by row) and computes the sum of each row and the sum of each column, printing the results in a labelled table.

8.8 A company has 3 warehouses, 10 aisles in each warehouse, and 20 shelves in each aisle. Write a program which calculates the total inventory. To initialize the stock, use the `Rand` module introduced in chapter 7. You will need a 3 dimensional array `stock` which should be initialized by the statement:

```
stock[w, a, s] := Rand.Int(25, 100)
```

within a nested `FOR` loop. Find the total inventory and print it out.

Chapter 9 — Characters & Strings

Numbers are not the only kind of data a program needs to work with. Names, words, sentences, commands, labels — all of these are text, and text is everywhere in real programs. This chapter is about how Oberon-07 represents and manipulates text: first at the level of individual characters, then at the level of strings — sequences of characters treated as a single unit.

We have already used both `CHAR` and strings in passing, in `Out.Char` and `Out.String`. Now we will examine them properly, including the important question of how strings are stored in memory, how to process them character by character, and how to use the `Strings` module that comes with OBNC.

9.1 The CHAR Type

A value of type `CHAR` holds a single character. Characters in Oberon-07 are drawn from a character set in which each character is assigned a unique integer code called its **ordinal value**. The character set used by OBNC is **ASCII**, which stands for the *American Standard Code for Information Interchange*. It is a character encoding standard that assigns a unique integer — an ordinal value — to each printable character and a set of non-printable control characters. ASCII defines 128 characters in total: ordinals 0 to 31 are control characters (non-printable, such as newline and tab), ordinals 32 to 126 are printable characters (letters, digits, punctuation, and the space character), and ordinal 127 is a special delete character. OBNC uses ASCII as its character set, so the ordinal value of any character in an Oberon-07 program is its ASCII code. A full ASCII table is provided in Appendix D for reference, but here are a few examples:

Character	Ordinal value
`"A"`	65
`"B"`	66
`"Z"`	90
`"a"`	97
`"z"`	122
`"0"`	48
`"9"`	57
`" "`	32 (space)

Character literals in Oberon-07 are written in double quotation marks and must contain exactly one character: `"A"`, `"z"`, `"?"`, `"3"`. Single characters can also be written as a two-digit hexadecimal ordinal followed by `X`: `41X` is the same as `"A"` (since 41 in hexadecimal is 65 in decimal). In practice, the quoted form is almost always clearer.

A `CHAR` variable is declared and assigned in the usual way:

```
VAR
  ch : CHAR;
...
ch := "A";
```

Comparing Characters

Because every character has an ordinal value, characters can be compared with the standard relational operators. The comparison is based on ordinal value, so:

```
"A" < "B"    (* TRUE: 65 < 66 *)
"a" > "Z"    (* TRUE: 97 > 90 — lowercase letters come after uppercase *)
"0" < "9"    (* TRUE: 48 < 57 *)
```

This makes it easy to test whether a character falls within a particular range:

```
IF (ch >= "A") & (ch <= "Z") THEN
  Out.String("uppercase letter")
END
```

The three ranges you will use most often are `"A"` to `"Z"` (uppercase letters), `"a"` to `"z"` (lowercase letters), and `"0"` to `"9"` (digit characters). These ranges are contiguous in ASCII, so a single range test is all that is needed.

9.2 ORD and CHR

Two predefined functions convert between characters and their ordinal values.

`ORD(ch)` returns the ordinal value of character `ch` as an integer:

```
ORD("A")    (* result: 65 *)
ORD("a")    (* result: 97 *)
ORD("0")    (* result: 48 *)
```

`CHR(n)` returns the character whose ordinal value is the integer `n`:

```
CHR(65)    (* result: "A" *)
CHR(97)    (* result: "a" *)
CHR(48)    (* result: "0" *)
```

`CHR` and `ORD` are inverses of each other: `CHR(ORD(ch)) = ch` for any character `ch`, and `ORD(CHR(n)) = n` for any valid ordinal value `n`.

These functions are useful for character arithmetic. For example, the uppercase letters run from ordinal 65 (`"A"`) to 90 (`"Z"`), and the corresponding lowercase letters run from 97 (`"a"`) to 122 (`"z"`).

The difference between the ordinal of a lowercase letter and its uppercase equivalent is always 32. This means we can convert a letter to lowercase with:

```
lower := CHR(ORD(upper) + 32)
```

and to uppercase with:

```
upper := CHR(ORD(lower) - 32)
```

More robustly, we can find the position of a letter in the alphabet regardless of case. If `ch` is an uppercase letter, `ORD(ch) - ORD("A")` gives its zero-based position (0 for `"A"`, 1 for `"B"`, and so on). Similarly, digit characters can be converted to their numeric values: `ORD(ch) - ORD("0")` gives the integer value of the digit character `ch`.

Example 9.1: Character Classification

```
MODULE CharClass;
  IMPORT In, Out;
VAR
  ch : CHAR;
BEGIN
  Out.String("Enter a character: ");
  Out.Ln;
  In.Char(ch);
  Out.String("Character: ");
  Out.Char(ch);
  Out.Ln;
  Out.String("Ordinal:   ");
  Out.Int(ORD(ch), 0);
  Out.Ln;
  IF (ch >= "A") & (ch <= "Z") THEN
    Out.String("Type: uppercase letter")
  ELSIF (ch >= "a") & (ch <= "z") THEN
    Out.String("Type: lowercase letter")
  ELSIF (ch >= "0") & (ch <= "9") THEN
    Out.String("Type: digit")
  ELSE
    Out.String("Type: other")
  END;
  Out.Ln
END CharClass.
```

With input `"G"`:

```
Character: G
Ordinal:   71
Type: uppercase letter
```

9.3 User-Defined Types

Before we move on to strings, we need to introduce an important feature of Oberon-07 that will make string handling work correctly: the ability to give a type a name of your own choosing.

So far, the types we have used — `INTEGER`, `REAL`, `BOOLEAN`, `CHAR`, and `ARRAY n OF type` — are either built in to the language or described inline in a `VAR` declaration. Oberon-07 also allows you to define **named types** in a `TYPE` section, which appears between the `CONST` and `VAR` sections:

```
TYPE
  name = type description;
```

For example:

```
TYPE
  String20 = ARRAY 20 OF CHAR;
  Score    = INTEGER;
```

Once declared, a named type can be used anywhere a type is expected:

```
VAR
  firstName : String20;
  lastName  : String20;
  total     : Score;
```

This looks like a convenience, but it is more than that. Oberon-07 uses **name equivalence** for type checking: two variables are considered to have the same type only if they are declared using the same type name (or are both the same built-in type). The structure of the type alone is not enough.

This matters whenever you try to assign one variable to another. Consider:

```
VAR
  a : ARRAY 20 OF CHAR;
  b : ARRAY 20 OF CHAR;
...
a := b;    (* ERROR: incompatible types *)
```

Even though `a` and `b` are both arrays of 20 characters, the compiler sees two distinct, anonymous types and refuses the assignment. The fix is to give the type a name and declare both variables using

it:

```
TYPE
  String20 = ARRAY 20 OF CHAR;
VAR
  a : String20;
  b : String20;
...
a := b;    (* OK: both are of type String20 *)
```

Now the compiler sees that `a` and `b` share the same named type and allows the assignment.

This is Oberon-07's strictness working in your favour. Name equivalence prevents you from accidentally mixing up arrays that happen to have the same size but represent different things — for example, a 20-character name and a 20-character address. By giving them different type names, the compiler will stop you from assigning one to the other even if their structures are identical.

The `TYPE` section follows the same ordering rules as other declaration sections: `CONST` first, then `TYPE`, then `VAR`. A type name may be used in `VAR` declarations that follow it, and in procedure declarations (covered in Part III).

We will use named types throughout the rest of this chapter and the chapters that follow. They are particularly important for strings, records (Chapter 10), and pointers (Chapter 11).

9.4 Strings in Oberon-07

A **string** in Oberon-07 is a sequence of characters stored in an array of type `ARRAY n OF CHAR`, terminated by a special character: the **null character**, written `0X` (ordinal value 0). The null terminator marks the end of the string within the array, so the array can be larger than the string it currently holds.

This representation — sometimes called a **null-terminated string** — is important to understand. If you declare:

```
TYPE
  String20 = ARRAY 20 OF CHAR;
VAR
  name : String20;
```

then `name` can hold a string of up to 19 characters plus the null terminator. The array has 20 positions (indices 0 to 19); the string occupies some prefix of those positions, with `0X` immediately following the last character. Positions after the null terminator are irrelevant.

For example, if `name` contains the string `"Alice"`, the array looks like this:

```
Index:  0    1    2    3    4    5    6  ...  19
Value: "A"  "l"  "i"  "c"  "e"  0X   ?  ...   ?
```

The ? entries are irrelevant — only positions 0 to 5 matter.

Assigning Strings

A string literal can be assigned directly to a character array, provided the array is large enough to hold all the characters plus the null terminator. The compiler inserts `0X` automatically:

```
name := "Alice";    (* stores "A","l","i","c","e",0X in name[0..5] *)
```

If the string literal is longer than the array can hold (including the terminator), the compiler will report an error. You can also assign one string variable to another, provided both are declared with the same named type:

```
TYPE
  String20 = ARRAY 20 OF CHAR;
VAR
  a : String20;
  b : String20;
...
a := "Hello";
b := a;          (* b now contains "Hello" *)
```

String Literals in Procedures

We have already been passing string literals to `Out.String` since Chapter 1. We can also pass string variables:

```
name := "Alice";
Out.String(name);    (* prints: Alice *)
```

`Out.String` prints characters from the array up to (but not including) the first `0X`.

9.5 Processing Strings Character by Character

Because a string is just an array of `CHAR` with a null terminator, you can process it character by character using a loop. The standard pattern is to step through the array until `0X` is encountered:

```
i := 0;
WHILE name[i] # 0X DO
  (* process name[i] *)
  INC(i)
END
```

This idiom appears constantly in string processing code. The loop condition `name[i] # 0X` is the standard way to ask "have we reached the end of the string?" It works because every valid string is

guaranteed to contain a `0X` terminator somewhere within the array, so the loop will always stop. Note that the loop variable `i` will equal the length of the string when the loop ends — the same value that `Strings.Length` would return — which can be useful if you need to know where the string ends after processing it.

Example 9.2: Counting Vowels

```
MODULE Vowels;
  IMPORT In, Out;
CONST
  MaxLen = 100;
TYPE
  Line = ARRAY MaxLen OF CHAR;
VAR
  sentence : Line;
  i, count : INTEGER;
  ch       : CHAR;
BEGIN
  Out.String("Enter a sentence: ");
  Out.Ln;
  In.Line(sentence);
  count := 0;
  i     := 0;
  WHILE sentence[i] # 0X DO
    ch := sentence[i];
    IF (ch = "a") OR (ch = "e") OR (ch = "i") OR
       (ch = "o") OR (ch = "u") OR (ch = "A") OR
       (ch = "E") OR (ch = "I") OR (ch = "O") OR
       (ch = "U") THEN
      INC(count)
    END;
    INC(i)
  END;
  Out.String("Vowels: ");
  Out.Int(count, 0);
  Out.Ln
END Vowels.
```

With input `"Hello, World!"`:

```
Vowels: 3
```

Note the use of `In.Line` to read a complete line of text including spaces. As discussed in Chapter 4, `In.Name` reads only up to the first whitespace, which would not work for a full sentence.

9.6 The Strings Module

Processing strings character by character with loops works, but common operations — finding the length of a string, searching for a pattern, appending one string to another — would require a lot of repetitive code if written from scratch every time. The `Strings` module provides ready-made procedures for these tasks.

To use it, add `Strings` to your import list:

```
IMPORT Strings;
```

Here is a complete reference to the module's procedures.

Strings.Length

```
Strings.Length(s)
```

Returns the number of characters in `s` up to and excluding the first `0X`. This is the length of the string, not the size of the array. For example:

```
name := "Alice";
n := Strings.Length(name);    (* n = 5 *)
```

Strings.Append

```
Strings.Append(extra, dest)
```

Appends the string `extra` to the end of the string `dest`. The result is stored in `dest`. If `dest` is not large enough to hold the combined result, the output is truncated but always properly terminated with `0X`.

```
greeting := "Hello, ";
Strings.Append("World!", greeting);
Out.String(greeting);    (* prints: Hello, World! *)
```

Strings.Insert

```
Strings.Insert(source, pos, dest)
```

Inserts the string `source` into `dest` at position `pos` (zero-based). If `pos` equals the length of `dest`, the effect is the same as `Append`. The result is truncated if `dest` is not large enough.

```
s := "Helo";
Strings.Insert("l", 2, s);
Out.String(s);    (* prints: Hello *)
```

Strings.Delete

```
Strings.Delete(s, pos, n)
```

Deletes `n` characters from `s` starting at position `pos`. If `n` is greater than the number of characters from `pos` to the end, everything from `pos` onwards is deleted.

```
s := "Hello, World!";
Strings.Delete(s, 5, 7);
Out.String(s);    (* prints: Hello! *)
```

Strings.Replace

```
Strings.Replace(source, pos, dest)
```

Replaces characters in `dest` starting at position `pos` with the string `source`. This has the same effect as deleting `Strings.Length(source)` characters at `pos` and then inserting `source` at `pos`.

```
s := "Hello, World!";
Strings.Replace("Oberon", 7, s);
Out.String(s);    (* prints: Hello, Oberon *)
```

Strings.Extract

```
Strings.Extract(source, pos, n, dest)
```

Extracts `n` characters from `source` starting at position `pos` and stores them in `dest`. If there are fewer than `n` characters available from `pos`, only the available characters are extracted.

```
s := "Hello, World!";
Strings.Extract(s, 7, 5, sub);
Out.String(sub);    (* prints: World *)
```

Strings.Pos

```
Strings.Pos(pattern, s, pos)
```

Searches for the first occurrence of `pattern` in `s`, starting the search at position `pos`. Returns the position of the match if found, or `-1` if not found.

```
s := "Hello, World!";
p := Strings.Pos("World", s, 0);    (* p = 7  *)
p := Strings.Pos("xyz",   s, 0);    (* p = -1 *)
```

Strings.Cap

```
Strings.Cap(s)
```

Converts every lowercase letter in `s` to its uppercase equivalent. Other characters are left unchanged.

```
s := "Hello, World!";
Strings.Cap(s);
Out.String(s);    (* prints: HELLO, WORLD! *)
```

Example 9.3: Using the Strings Module

This program reads a word from the user, reports its length, converts it to uppercase, and searches for the letter `"e"` within it:

```
MODULE StringDemo;
  IMPORT In, Out, Strings;
CONST
  MaxLen = 50;
TYPE
  String50 = ARRAY MaxLen OF CHAR;
VAR
  word  : String50;
  upper : String50;
  pos   : INTEGER;
BEGIN
  Out.String("Enter a word: ");
  Out.Ln;
  In.Name(word);
  Out.String("You entered:  ");
  Out.String(word);
  Out.Ln;
```

```
  Out.String("Length:       ");
  Out.Int(Strings.Length(word), 0);
  Out.Ln;
  upper := word;
  Strings.Cap(upper);
  Out.String("Uppercase:    ");
  Out.String(upper);
  Out.Ln;
  pos := Strings.Pos("e", word, 0);
  IF pos >= 0 THEN
    Out.String("First 'e' at position: ");
    Out.Int(pos, 0)
  ELSE
    Out.String("No 'e' found")
  END;
  Out.Ln
END StringDemo.
```

With input `"oberon"`:

```
You entered:  oberon
Length:       6
Uppercase:    OBERON
First 'e' at position: 2
```

Because `word` and `upper` are both declared as `String50`, the assignment `upper := word` is valid — they share the same named type.

9.7 Arrays of Strings

Since a string in Oberon-07 is an array of `CHAR`, an array of strings is a two-dimensional array of `CHAR`. The cleanest way to declare one is to first define a named type for the string, then declare an array of that type:

```
CONST
  MaxWordLen = 20;
  ListSize   = 5;
TYPE
  Word = ARRAY MaxWordLen OF CHAR;
VAR
  names : ARRAY ListSize OF Word;
```

This declares `names` as an array of 5 elements, where each element is a `Word` — an `ARRAY 20 OF CHAR`. Each element `names[i]` is a complete string variable and can be used wherever a string variable can:

```
names[0] := "Alice";
names[1] := "Bob";
Out.String(names[0]);                        (* prints: Alice *)
Out.Int(Strings.Length(names[1]), 0);        (* prints: 3     *)
```

And because each element has the named type `Word`, you can assign one element to another or to any other `Word` variable without a type error:

```
VAR
  temp : Word;
...
temp     := names[0];    (* OK: both are of type Word *)
names[0] := names[1];
names[1] := temp;
```

Example 9.4: A Word List

This program stores a small list of words, reads a word from the user, and checks whether it appears in the list:

```
MODULE WordList;
  IMPORT In, Out;
CONST
  ListSize   = 5;
  MaxWordLen = 20;
TYPE
  Word = ARRAY MaxWordLen OF CHAR;
VAR
  list  : ARRAY ListSize OF Word;
  word  : Word;
  i     : INTEGER;
  found : BOOLEAN;
BEGIN
  list[0] := "apple";
  list[1] := "banana";
  list[2] := "cherry";
  list[3] := "date";
  list[4] := "elderberry";
  Out.String("Enter a word to search for: ");
  Out.Ln;
  In.Name(word);
  found := FALSE;
  i     := 0;
  WHILE (i < ListSize) & ~found DO
    IF list[i] = word THEN
```

```
      found := TRUE
    END;
    INC(i)
  END;
  IF found THEN
    Out.String("'");
    Out.String(word);
    Out.String("' was found in the list.")
  ELSE
    Out.String("'");
    Out.String(word);
    Out.String("' was not found in the list.")
  END;
  Out.Ln
END WordList.
```

With input `"cherry"`:

```
'cherry' was found in the list.
```

With input `"fig"`:

```
'fig' was not found in the list.
```

The string comparison `list[i] = word` uses the ordinary equality operator `=`. In Oberon-07, two character arrays of the same named type can be compared with `=` and `#` directly — the comparison checks whether the string contents are identical. The loop condition `(i < ListSize) & ~found` stops the search as soon as a match is found, avoiding unnecessary comparisons.

9.8 Sorting a Word List

Now that we have an array of strings, let us apply a classic algorithm to it: **sorting**. Sorting means arranging the elements of an array into order — alphabetical order for strings, ascending or descending order for numbers. It is one of the most fundamental operations in computing, and one that beginners often find difficult at first. We will therefore develop the algorithm carefully, using the stepwise refinement technique from Chapter 7, before writing a single line of code.

The algorithm we will use is called **selection sort**. It is not the fastest sorting algorithm, but it is one of the easiest to understand, and understanding it well is a better foundation than knowing a faster algorithm poorly.

Developing the Algorithm

First refinement:

```
Sort the array into alphabetical order.
```

Not yet useful. Let us think about what sorting actually means in practice.

Second refinement:

```
Repeatedly find the smallest remaining word and move it into place.
```

This is the core idea. Imagine the array is divided into two parts: a sorted portion at the left, and an unsorted portion at the right. At the start, the sorted portion is empty and the whole array is unsorted. On each pass, we find the smallest word in the unsorted portion and move it to the end of the sorted portion. When the unsorted portion is empty, the whole array is sorted.

Third refinement:

```
For each position i from 0 to N-2:
    Find the position of the smallest word from position i to N-1.
    Swap that word with the word at position i.
```

Let us trace through a concrete example to make sure this is right. Suppose we have four words:

```
Position:  0       1       2       3
Content:  "plum"  "apple"  "mango"  "banana"
```

Pass 1 (i = 0): We scan positions 0 to 3. The smallest word is `"apple"` at position 1. We swap positions 0 and 1:

```
Position:  0        1       2       3
Content:  "apple"  "plum"  "mango"  "banana"
```

The sorted portion now contains `"apple"`. The unsorted portion is positions 1 to 3.

Pass 2 (i = 1): We scan positions 1 to 3. The smallest word is `"banana"` at position 3. We swap positions 1 and 3:

```
Position:  0        1         2       3
Content:  "apple"  "banana"  "mango"  "plum"
```

Pass 3 (i = 2): We scan positions 2 to 3. The smallest word is `"mango"` at position 2 — already in place. No swap needed.

The array is now fully sorted. Notice that we only needed N-1 passes: once N-1 elements are in their correct positions, the last element must be correct too. This is why the outer loop runs from `0` to `N - 2`, not `N - 1`.

Fourth refinement — finding the minimum position:

The step "find the position of the smallest word from position i to N-1" needs more detail. We cannot find a minimum by looking at the whole range at once; we must scan it element by element, keeping track of the position of the smallest word seen so far:

```
Set minPos to i.
For each position j from i+1 to N-1:
    If words[j] < words[minPos]:
        Set minPos to j.
```

After this inner loop, `minPos` holds the index of the smallest word in the unsorted portion. If `minPos` is still `i`, the first element was already the smallest and no swap is needed.

Fifth refinement — performing the swap:

Swapping two string variables requires a temporary variable, just as swapping two integers does. We cannot write `words[i] := words[minPos]` and then `words[minPos] := words[i]` — the first statement would overwrite `words[i]` before we had saved its value. Instead:

```
Copy words[i] into temp.
Copy words[minPos] into words[i].
Copy temp into words[minPos].
```

We now have enough detail to write the code directly.

The Program

```
MODULE SortWords;
  IMPORT Out;
CONST
  N          = 6;
  MaxWordLen = 20;
TYPE
  Word = ARRAY MaxWordLen OF CHAR;
VAR
  words        : ARRAY N OF Word;
  temp         : Word;
  i, j, minPos : INTEGER;
BEGIN
  words[0] := "plum";
  words[1] := "apple";
  words[2] := "mango";
  words[3] := "banana";
  words[4] := "cherry";
  words[5] := "date";
  (* Selection sort *)
  FOR i := 0 TO N - 2 DO
    minPos := i;
    FOR j := i + 1 TO N - 1 DO
      IF words[j] < words[minPos] THEN
        minPos := j
```

```
      END
    END;
    IF minPos # i THEN
      temp          := words[i];
      words[i]      := words[minPos];
      words[minPos] := temp
    END
  END;
  Out.String("Sorted list:");
  Out.Ln;
  FOR i := 0 TO N - 1 DO
    Out.String("  ");
    Out.String(words[i]);
    Out.Ln
  END
END SortWords.
```

Output:

```
Sorted list:
  apple
  banana
  cherry
  date
  mango
  plum
```

The comparison `words[j] < words[minPos]` works correctly because Oberon-07's character array comparison is **lexicographic** — it compares strings the same way a dictionary would, character by character from left to right. This means the same sorting code works equally well for any array of `Word` values.

Because `temp`, `words[i]`, and `words[minPos]` are all of the named type `Word`, the swap assignments are valid. This is exactly the situation where named types pay off: the compiler can verify that we are swapping values of genuinely the same type, and the swap itself is expressed as three simple assignment statements rather than a character-by-character loop.

9.9 String Sizes

One practical issue with strings in Oberon-07 is that the size of a character array must be declared in advance and cannot be changed at runtime. This means you must decide, when writing the program, how long any given string might need to be, and declare the array large enough to hold it.

The consequences of declaring an array that is too small depend on where the **overflow** occurs (an overflow refers to a situation when data exceeds the storage capacity of a system or memory space). String literals assigned at compile time will cause a compiler error if they are too long — a helpful early warning. Strings built up at runtime using `Strings.Append` or `In.Line` will be silently truncated

to fit, with `0X` always written at the end to ensure the string remains valid. This truncation can cause subtle bugs if you are not aware of it.

The safe habits are: declare string arrays generously, use named constants for the size (so it is easy to change), and be aware that `Strings` procedures will truncate rather than overflow when the destination is too small.

A Note on using In.Name with In.Done

Exercises 9.6 and 9.7 asks you to write programs which prompts the user to enter words, stopping when `In.Done` is false. The conditions which make it false are worth examining carefully. `In.Name` reads a sequence of non-whitespace characters. `In.Done` becomes `FALSE` when `In.Name` reaches the end of the input stream without finding any non-whitespace characters to read. In practice, when running a program interactively at the terminal, this means pressing `Ctrl+D` (on Linux/macOS), which sends an end-of-file signal to the program's standard input.

Simply pressing `Enter` on a blank line will not make `In.Done` false — `In.Name` skips all whitespace including newlines while looking for the next word, so a blank line is just ignored and the program keeps waiting. This is different from `In.Line`, which returns an empty string on a blank line.

So for exercises 9.6 and 9.7 the user would type words one per line (or separated by spaces if typed on one line) and press Ctrl+D when finished.

Summary

- `CHAR` holds a single character. Characters are ordered by their **ordinal value** (ASCII code). The relational operators work on `CHAR` by ordinal value.
- `ORD(ch)` returns the integer ordinal of character `ch`. `CHR(n)` returns the character with ordinal `n`. They are inverse functions.
- Oberon-07 uses **name equivalence** for type checking: two variables are compatible only if they share the same named type, not merely the same structure. Use a `TYPE` declaration to give a type a name.
- The `TYPE` section appears between `CONST` and `VAR` in the declaration order.
- A **string** is a null-terminated sequence of characters stored in an `ARRAY n OF CHAR`. The null terminator `0X` marks the end of the string within the array.
- A string literal can be assigned to a character array provided the array is large enough (at least `length + 1` for the terminator).
- The standard idiom for iterating over a string is `WHILE s[i] # 0X DO ... INC(i) END`.
- Two character array variables of the same named type can be compared with `=` and `#`; the comparison is lexicographic.
- The `Strings` module provides: `Length` (string length), `Append` (append one string to another), `Insert` (insert at a position), `Delete` (remove characters), `Replace` (overwrite at a position), `Extract` (copy a substring), `Pos` (find a pattern), and `Cap` (convert to uppercase).
- An **array of strings** is best declared by first defining a named type for the string (`TYPE Word = ARRAY n OF CHAR`) and then declaring an array of that type. This ensures all elements share the same type name and can be freely assigned to one another and to temporary variables of the same type.
- **Selection sort** works by repeatedly finding the smallest element in the unsorted portion of the array and swapping it into place. The outer loop runs from 0 to N-2; the inner loop scans

the unsorted portion to find the minimum position.

Exercises

9.1 Write a program that reads a character from the user and prints its ordinal value. If the character is a lowercase letter, also print the corresponding uppercase letter using `CHR` and `ORD`.

9.2 Write a program that reads a word using `In.Name` and prints it reversed. For example, `"oberon"` should produce `"norebo"`. (Hint: use `Strings.Length` to find the last character's index, then loop backwards printing each character with `Out.Char`.)

9.3 Write a program that reads a sentence using `In.Line` and counts how many words it contains.

9.4 Write a program that reads a string and checks whether it is a palindrome — that is, whether it reads the same forwards and backwards. For example, `"racecar"` is a palindrome but `"oberon"` is not.

9.5 Write a program that declares an array of 7 strings representing the days of the week (`"Monday"` through `"Sunday"`), reads a number from 1 to 7 from the user, and prints the corresponding day name. Use array indexing to select the name directly.

9.6 Write a program that reads up to 10 words on one line (using `In.Name`, stopping when `In.Done` is false) and stores them in an array of strings. Print the list back in the order entered, then print the longest word. Use `Strings.Length` to compare lengths.

9.7 Extend the `SortWords` program from Section 9.8 to read its word list from the user rather than having it fixed in the source code. Read up to `N` words using `In.Name`, stopping when `In.Done` is false or the array is full, then sort and print the result.

9.8 Write a program that reads a sentence and reports the position of the first occurrence of each vowel (`a`, `e`, `i`, `o`, `u`) using `Strings.Pos`. If a vowel does not appear in the sentence, print a suitable message for that vowel. Make sure your program works correctly for uppercase vowels in the sentence.

Chapter 10 — Records

Arrays, which we studied in Chapter 8, are ideal for collections of values that are all the same type. But many real-world things are naturally described by a collection of values of *different* types. A contact in an address book has a name (text), a phone number (text), and an age (a number). A date has a day, a month, and a year (all integers, but logically distinct). A product in a shop has a name (text), a price (a real number), and a stock count (an integer).

In all these cases, the different pieces of information belong together — they describe a single thing — yet they are of different types and so cannot be stored in an array. What we need is a way to group related data of different types into a single named unit. In Oberon-07, this is the **record**.

10.1 What Is a Record?

A record is a data structure that groups a fixed set of named values, called **fields**, into a single unit. Each field has its own name and its own type, independently of the other fields. Once declared, a record variable holds all its fields together and can be passed around, assigned, and stored as a single entity.

Think of a record as a form with labelled boxes. A date form might have three boxes: one labelled `day`, one labelled `month`, and one labelled `year`. Filling in the form means giving each box a value. A record variable is exactly this: a named collection of labelled boxes, each capable of holding one value of its declared type.

10.2 Declaring Record Types

Records are always introduced through a `TYPE` declaration. (As we saw in Chapter 9, Oberon-07's name equivalence makes named types essential whenever you want to assign records to one another or pass them to procedures.) The syntax is:

```
TYPE
  TypeName = RECORD
    fieldName : fieldType;
    fieldName : fieldType;
    ...
  END;
```

Fields of the same type can be grouped on a single line:

```
TYPE
  Date = RECORD
    day, month, year : INTEGER
  END;
```

Here `Date` is a record type with three integer fields. A variable of type `Date` holds all three values together:

```
VAR
  today    : Date;
  birthday : Date;
```

More complex records can mix field types freely:

```
TYPE
  NameStr = ARRAY 30 OF CHAR;
  Contact = RECORD
    name  : NameStr;
    age   : INTEGER;
    score : REAL
  END;
```

Notice that we first define `NameStr` as a named type for the string, following the lesson from Chapter 9. This is the standard pattern: string fields in records should use named string types so that they can be assigned freely.

10.3 Accessing Fields

Individual fields of a record are accessed using the **dot selector** — the record variable name, a dot, and the field name:

```
today.day   := 15;
today.month := 6;
today.year  := 2025;
```

A field access can appear anywhere a variable of the field's type can appear — on the left of an assignment, in an expression, or as an argument to a procedure:

```
Out.Int(today.day, 0);
Out.String(contact.name);
age := person.age + 1;
```

The dot selector can be chained. If a record contains a field that is itself a record, you use two dots:

```
TYPE
  Person = RECORD
    name      : NameStr;
    birthdate : Date
  END;
VAR
  p : Person;
...
```

```
p.birthdate.year := 1990;
Out.Int(p.birthdate.day, 0);
```

Example 10.1: A Date Record

```
MODULE DateDemo;
  IMPORT Out;
TYPE
  Date = RECORD
    day, month, year : INTEGER
  END;
VAR
  d : Date;
BEGIN
  d.day   := 15;
  d.month := 6;
  d.year  := 2025;
  Out.Int(d.day, 0);
  Out.String("/");
  Out.Int(d.month, 0);
  Out.String("/");
  Out.Int(d.year, 0);
  Out.Ln
END DateDemo.
```

Output:

```
15/6/2025
```

10.4 Assigning Records

A record variable can be assigned to another variable of the same type with a single assignment statement. The assignment copies all fields:

```
VAR
  a, b : Date;
...
a.day   := 1;
a.month := 1;
a.year  := 2000;
b := a;    (* b now has day=1, month=1, year=2000 *)
```

After this assignment, `a` and `b` hold the same values but are independent copies — modifying a field

of `b` afterwards does not affect `a`. Both variables must be of the same named record type for the assignment to be valid.

10.5 Arrays Inside Records

Just as a record can contain fields of any type, a field can itself be an array. This is useful whenever a single record needs to hold a collection of related values — a student's scores across several tests, for example, or a week's worth of temperature readings.

The declaration is straightforward: the field is simply declared with an array type, using a named type as always:

```
CONST
  NumTests = 4;
TYPE
  Scores  = ARRAY NumTests OF INTEGER;
  Student = RECORD
    name   : ARRAY 20 OF CHAR;
    scores : Scores
  END;
```

Accessing an element of the array field combines the dot selector with an array index:

```
VAR
  s : Student;
...
s.name      := "Alice";
s.scores[0] := 78;
s.scores[1] := 85;
s.scores[2] := 91;
s.scores[3] := 73;
```

The expression `s.scores[2]` reads as "the third element of the `scores` field of record `s`". This notation extends naturally to loops:

```
total := 0;
FOR i := 0 TO NumTests - 1 DO
  INC(total, s.scores[i])
END;
mean := FLT(total) / FLT(NumTests);
```

Example 10.2: Test Scores

This program stores a student record with an array of test scores and prints the scores along with their average:

```
MODULE TestScores;
  IMPORT Out;
CONST
  NumTests = 4;
  NameLen  = 20;
TYPE
  NameStr = ARRAY NameLen OF CHAR;
  Scores  = ARRAY NumTests OF INTEGER;
  Student = RECORD
    name   : NameStr;
    scores : Scores
  END;
VAR
  s     : Student;
  i     : INTEGER;
  total : INTEGER;
BEGIN
  s.name      := "Alice";
  s.scores[0] := 78;
  s.scores[1] := 85;
  s.scores[2] := 91;
  s.scores[3] := 73;
  Out.String("Student: ");
  Out.String(s.name);
  Out.Ln;
  Out.String("Scores:  ");
  total := 0;
  FOR i := 0 TO NumTests - 1 DO
    Out.Int(s.scores[i], 0);
    IF i < NumTests - 1 THEN Out.String(", ") END;
    INC(total, s.scores[i])
  END;
  Out.Ln;
  Out.String("Average: ");
  Out.Real(FLT(total) / FLT(NumTests), 0);
  Out.Ln
END TestScores.
```

Output:

```
Student: Alice
Scores:  78, 85, 91, 73
Average: 8.17500E+01
```

10.6 Records and Input

Reading values into a record is simply a matter of reading into each field in turn. There is no single procedure that reads an entire record at once; you read field by field:

```
Out.String("Day:   "); Out.Ln; In.Int(d.day);
Out.String("Month: "); Out.Ln; In.Int(d.month);
Out.String("Year:  "); Out.Ln; In.Int(d.year);
```

This is straightforward but does involve some repetition. In Part III, when we introduce procedures, we will see how to wrap this kind of repeated pattern into a reusable block.

10.7 Arrays of Records

One of the most useful combinations in all of programming is an array of records. This is the natural representation for a list of structured items: a class list (an array of student records), a product catalogue (an array of product records), a contact book (an array of contact records).

The declaration follows the same pattern as any array: first define the record type, then declare an array of that type:

```
CONST
  ClassSize = 5;
  NameLen   = 20;
TYPE
  NameStr = ARRAY NameLen OF CHAR;
  Student = RECORD
    name  : NameStr;
    score : INTEGER
  END;
VAR
  class : ARRAY ClassSize OF Student;
```

Each element `class[i]` is a complete `Student` record. To access a field of a particular element, combine the array index and the dot selector:

```
class[0].name  := "Alice";
class[0].score := 87;
class[1].name  := "Bob";
class[1].score := 74;
```

This combines naturally with `FOR` loops:

```
FOR i := 0 TO ClassSize - 1 DO
  Out.String(class[i].name);
  Out.String(": ");
```

```
  Out.Int(class[i].score, 0);
  Out.Ln
END
```

Example 10.3: Class Scores

This program stores a small class list with names and scores, then prints the list and reports the highest score:

```
MODULE ClassScores;
  IMPORT Out, Strings;
CONST
  ClassSize = 4;
  NameLen   = 15;
  ColWidth  = 15;
TYPE
  NameStr = ARRAY NameLen OF CHAR;
  Student = RECORD
    name  : NameStr;
    score : INTEGER
  END;
VAR
  class : ARRAY ClassSize OF Student;
  best  : INTEGER;
  i, j  : INTEGER;
BEGIN
  class[0].name := "Alice";  class[0].score := 87;
  class[1].name := "Bob";    class[1].score := 74;
  class[2].name := "Carol";  class[2].score := 91;
  class[3].name := "David";  class[3].score := 68;
  Out.String("Name         Score");
  Out.Ln;
  Out.String("----         -----");
  Out.Ln;
  FOR i := 0 TO ClassSize - 1 DO
    Out.String(class[i].name);
    j := Strings.Length(class[i].name);
    WHILE j < ColWidth DO
      Out.Char(" ");
      INC(j)
    END;
    Out.Int(class[i].score, 0);
    Out.Ln
  END;
  best := class[0].score;
```

```
  FOR i := 1 TO ClassSize - 1 DO
    IF class[i].score > best THEN
      best := class[i].score
    END
  END;
  Out.Ln;
  Out.String("Highest score: ");
  Out.Int(best, 0);
  Out.Ln
END ClassScores.
```

Output:

```
Name         Score
----         -----
Alice          87
Bob            74
Carol          91
David          68

Highest score: 91
```

The column alignment is achieved by padding each name with spaces after printing it. `Strings.Length` gives the number of characters actually printed, and the `WHILE` loop then outputs spaces until the total reaches `ColWidth`. This ensures that the score always starts at the same horizontal position regardless of how long the name is. The constant `ColWidth` makes it easy to adjust the column width in one place if needed. We will meet more flexible formatting tools in Chapter 19, but this manual padding technique is useful to know and works in any situation.

10.8 Sorting an Array of Records

Sorting an array of records works in exactly the same way as sorting an array of strings (Chapter 9) — we use the same selection sort algorithm. The only difference is that the comparison is made on a particular field, and the swap exchanges whole records rather than individual values.

Suppose we want to sort the `class` array from the previous section into alphabetical order by name. The comparison `class[j].name < class[minPos].name` compares the name fields of two records lexicographically, exactly as in Chapter 9's word sort. The swap exchanges the entire records using a temporary variable of type `Student`:

Example 10.4

```
MODULE SortClass;
  IMPORT Out;
CONST
  ClassSize = 4;
```

```
  NameLen   = 20;
TYPE
  NameStr = ARRAY NameLen OF CHAR;
  Student = RECORD
    name  : NameStr;
    score : INTEGER
  END;
VAR
  class        : ARRAY ClassSize OF Student;
  temp         : Student;
  i, j, minPos : INTEGER;
BEGIN
  class[0].name := "David";  class[0].score := 68;
  class[1].name := "Alice";  class[1].score := 87;
  class[2].name := "Carol";  class[2].score := 91;
  class[3].name := "Bob";    class[3].score := 74;
  (* Selection sort by name *)
  FOR i := 0 TO ClassSize - 2 DO
    minPos := i;
    FOR j := i + 1 TO ClassSize - 1 DO
      IF class[j].name < class[minPos].name THEN
        minPos := j
      END
    END;
    IF minPos # i THEN
      temp        := class[i];
      class[i]    := class[minPos];
      class[minPos] := temp
    END
  END;
  Out.String("Sorted by name:");
  Out.Ln; Out.Ln;
  FOR i := 0 TO ClassSize - 1 DO
    Out.String("  ");
    Out.String(class[i].name);
    Out.String(": ");
    Out.Int(class[i].score, 0);
    Out.Ln
  END
END SortClass.
```

Output:

```
Sorted by name:
  Alice: 87
  Bob: 74
  Carol: 91
  David: 68
```

Two things are worth noting. First, when two records are swapped, all their fields move together — the name and the score stay paired, because we are swapping entire `Student` records rather than individual fields. This is one of the key benefits of using records: the data belonging to one entity travels together as a unit. Second, the `temp` variable must be of type `Student` — the same named type as the array elements — so that the record assignments in the swap are valid.

The same approach works for sorting by any field: to sort by score instead of name, simply change the comparison to `class[j].score < class[minPos].score`.

10.9 A Simple Contact Book

Let us bring everything together in a program that manages a small list of contacts. Each contact has a name, a phone number (stored as a string), and an age. The program reads the contacts from the user, prints them in a formatted list, and then prints them again sorted alphabetically by name. This is the most complex program in the book so far, so read it carefully, compile and run it, then read the explanatory notes below.

Example 10.5

```
MODULE Contacts;
  IMPORT In, Out;
CONST
  MaxContacts = 5;
  NameLen     = 25;
  PhoneLen    = 15;
TYPE
  NameStr  = ARRAY NameLen  OF CHAR;
  PhoneStr = ARRAY PhoneLen OF CHAR;
  Contact  = RECORD
    name  : NameStr;
    phone : PhoneStr;
    age   : INTEGER
  END;
VAR
  book         : ARRAY MaxContacts OF Contact;
  temp         : Contact;
  flush        : ARRAY 10 OF CHAR;
  n            : INTEGER;
  i, j, minPos : INTEGER;
```

```
  done         : BOOLEAN;
BEGIN
  n    := 0;
  done := FALSE;
  Out.String("Enter contacts (up to ");
  Out.Int(MaxContacts, 0);
  Out.String("). Press Enter with a blank name to finish.");
  Out.Ln;
  WHILE (n < MaxContacts) & ~done DO
    Out.String("Name:  ");
    In.Line(book[n].name);
    IF book[n].name[0] = 0X THEN
      done := TRUE
    ELSE
      Out.String("Phone: ");
      In.Name(book[n].phone);
      Out.String("Age:   ");
      In.Int(book[n].age);
      In.Line(flush);    (* consume leftover newline *)
      Out.Ln;
      INC(n)
    END
  END;
  IF n = 0 THEN
    Out.String("No contacts entered.");
    Out.Ln
  ELSE
    (* Print as entered *)
    Out.Ln;
    Out.String("Contact List");
    Out.Ln;
    Out.String("------------");
    Out.Ln;
    FOR i := 0 TO n - 1 DO
      Out.String("Name:  "); Out.String(book[i].name);  Out.Ln;
      Out.String("Phone: "); Out.String(book[i].phone); Out.Ln;
      Out.String("Age:   "); Out.Int(book[i].age, 0);   Out.Ln;
      Out.String("------------");
      Out.Ln
    END;
    (* Sort by name *)
    FOR i := 0 TO n - 2 DO
      minPos := i;
      FOR j := i + 1 TO n - 1 DO
        IF book[j].name < book[minPos].name THEN
          minPos := j
```

```
        END
      END;
      IF minPos # i THEN
        temp         := book[i];
        book[i]      := book[minPos];
        book[minPos] := temp
      END
    END;
    (* Print sorted *)
    Out.Ln;
    Out.String("Sorted alphabetically:");
    Out.Ln;
    Out.String("------------");
    Out.Ln;
    FOR i := 0 TO n - 1 DO
      Out.String("Name:  "); Out.String(book[i].name);  Out.Ln;
      Out.String("Phone: "); Out.String(book[i].phone); Out.Ln;
      Out.String("Age:   "); Out.Int(book[i].age, 0);   Out.Ln;
      Out.String("------------");
      Out.Ln
    END
  END
END Contacts.
```

A sample run with three contacts:

```
Enter contacts (up to 5). Press Enter with a blank name to finish.
Name:  Carol
Phone: 07700900789
Age:   31
Name:  Alice
Phone: 07700900123
Age:   34
Name:  Bob
Phone: 07700900456
Age:   28
Name:

Contact List
------------
Name:  Carol
Phone: 07700900789
Age:   31
------------
Name:  Alice
```

```
Phone: 07700900123
Age:   34
------------
Name:  Bob
Phone: 07700900456
Age:   28
------------

Sorted alphabetically:
------------
Name:  Alice
Phone: 07700900123
Age:   34
------------
Name:  Bob
Phone: 07700900456
Age:   28
------------
Name:  Carol
Phone: 07700900789
Age:   31
------------
```

The program uses `In.Line` rather than `In.Name` to read the contact's name. This is necessary for two reasons: `In.Line` reads the whole line including any spaces (allowing names like "Alice Smith"), and it returns an empty string when the user presses Enter on a blank line, which is how we detect that the user wants to stop. Checking `book[n].name[0] = 0X` tests whether the first character is the null terminator — the signature of an empty string.

Rather than forcing the loop to end by corrupting `n`, a separate `BOOLEAN` variable `done` is used to signal termination cleanly. This keeps `n` accurate as a count of contacts actually entered, which is important since it is used later to control the printing and sorting loops.

There is a subtlety involving the input buffer. After `In.Int` reads the age, the newline character left behind by the Enter keypress remains sitting in the input stream. On the next iteration, `In.Line` would read that leftover newline immediately and return an empty string — incorrectly signalling a blank name and terminating the loop one contact too early. The line:

```
In.Line(flush); (* consume leftover newline *)
```

handles this by reading and discarding whatever remains on the current line after the age — in normal use, just the newline itself. A dedicated `flush` variable is used for this purpose rather than reusing `book[n].name`, which would overwrite the name that was just stored. This is the same newline-flushing issue discussed in Chapter 4; whenever `In.Int` or `In.Real` is followed by `In.Line`, a dummy `In.Line` call is needed to clear the buffer first.

This program is a useful foundation. In Chapter 13 (Procedures) we will factor the input, printing, and sorting into separate reusable procedures; in Chapter 18 (Files) we will save and load the contact list from disk; and Chapter 22 brings all of these ideas together in a complete address book

application.

10.10 Records and the TYPE Section: A Summary Pattern

By now a clear pattern has emerged for how to organise declarations when records are involved. Here is the template:

```
CONST
  FieldSize = 30;       (* sizes for string fields *)
  MaxItems  = 100;      (* array capacity          *)
TYPE
  FieldStr = ARRAY FieldSize OF CHAR;    (* named string type first *)
  ItemRec  = RECORD                      (* then the record type    *)
    field1 : FieldStr;
    field2 : INTEGER;
    field3 : REAL
  END;
VAR
  items : ARRAY MaxItems OF ItemRec;     (* array of records        *)
  count : INTEGER;                       (* how many are in use     *)
  temp  : ItemRec;                       (* for swaps, copies, etc  *)
```

Following this pattern consistently will serve you well through the rest of the book. The key points are: string fields always use named array types; the record type is declared in `TYPE` after any string types it needs; and arrays of records are declared in `VAR` using the named record type. A `temp` variable of the record type is always worth declaring if you plan to sort or swap records.

Summary

- A **record** groups a fixed set of named **fields** of possibly different types into a single unit. It is declared in a `TYPE` section as `RECORD field : type; ... END`.
- Fields are accessed with the **dot selector**: `variable.fieldName`. Dot selectors can be chained for records within records.
- A field can itself be an array. Elements of such a field are accessed by combining the dot selector and an array index: `record.arrayField[i]`.
- Two record variables of the same named type can be assigned to one another with a single `:=` statement, which copies all fields.
- An **array of records** is the natural representation for a list of structured items. Individual elements and their fields are accessed with combined index and dot notation: `array[i].fieldName`.
- String fields in records should be declared using named string types to allow assignment and comparison.
- **Sorting an array of records** uses the same selection sort algorithm as sorting an array of strings or numbers. The comparison is made on a chosen field; the swap exchanges whole records using a temporary variable of the same record type.

Exercises

10.1 Declare a record type `Point` with two `REAL` fields, `x` and `y`. Write a program that reads two points from the user and prints the distance between them. (The distance between (x1, y1) and (x2, y2) is the square root of `(x2-x1)^2 + (y2-y1)^2`. The `Math` module provides `Math.sqrt`; you may use it here.)

10.2 Declare a record type `Date` with integer fields `day`, `month`, and `year`. Write a program that reads two dates from the user and prints which one is earlier. (Hint: compare year first, then month, then day.)

10.3 Declare a record type `Product` with a name (string), a price (`REAL`), and a stock count (`INTEGER`). Declare an array of 5 products, fill it with data in the source code, and print a formatted table showing all three fields for each product. Then print the name of the most expensive product.

10.4 Extend the `ClassScores` program from Section 10.7 so that after printing the table it also prints the name of the student with the highest score, not just the score itself.

10.5 Write a program that declares an array of up to 10 `Date` records, reads dates from the user (day, month, year on separate lines, stopping when `In.Done` is false) and prints all the dates entered in the format `DD/MM/YYYY`. Pad the day and month with a leading zero when they are less than 10. Remember that pressing Ctr-D terminates input (if you want to input less than 10 dates).

10.6 Declare a record type `Point` with `INTEGER` fields `x` and `y`. Then declare a record type `Rectangle` with fields `topLeft` and `bottomRight`, both of type `Point`. Write a program that reads two rectangles from the user (each defined by its top-left and bottom-right corners), and prints the width, height, and area of each. Then print which rectangle has the larger area. Access the nested fields as for example `rect.topLeft.x` and `rect.bottomRight.y`.

10.7 Using the `SortClass` program from Section 10.8 as a starting point, modify it to sort the class by score in descending order (highest score first) rather than by name. Print the sorted result.

10.8 Declare a record type `Rectangle` with `REAL` fields `width` and `height`. Write a program that reads an array of 4 rectangles from the user and prints the area and perimeter of each. Then sort the array by area in ascending order and print the sorted list.

Chapter 11 — Pointers & Dynamic Memory

Every variable we have used so far has been **static**: declared by name in a `VAR` section, given a fixed amount of memory by the compiler, and present for the entire lifetime of the program. This works well when you know at compile time how many things you need to store. But some problems are genuinely open-ended. How many contacts will the user enter? How many items are in the list? You cannot always know in advance.

This chapter introduces a different way of managing data: **dynamic memory allocation**. Instead of declaring a fixed number of variables at compile time, a program can request new memory at runtime, as many times as it needs, and connect that memory together into whatever shape the problem requires. The mechanism that makes this possible is the **pointer**.

Pointers are one of the most powerful ideas in programming — and one of the most confusing for beginners, because for the first time a variable's value is not the data you care about, but a way of *reaching* it. We will therefore go slowly.

Pointers are key to creating **Data Structures**. A data structure is a specialized format for organizing, processing, storing, and retrieving data efficiently in a computer's memory. Choosing the right data structure directly impacts the speed and memory efficiency of a program. You've already met two data structures in the book: arrays and records. Some other common data structures are:

- **Linked Lists**: A sequence of nodes where each node contains data and a way to access other nodes in the chain.
- **Stacks**: A Last-In, First-Out (LIFO) structure, like a stack of dinner plates where you add and remove from the top.
- **Queues**: A First-In, First-Out (FIFO) structure, like a queue in a grocery store.
- **Trees**: A hierarchical structure with a "root" node and parent-child connections, like a family tree or file folder system.

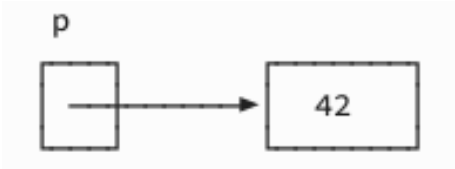
We won't be studying any of these in depth in the book. Instead, this chapter isolates the handful of primitive operations that pointers give you — allocate, dereference, link, traverse, redirect — and gives each one its own diagram. A linked list, a tree, a ring, a queue or a structure with no name at all are all just different patterns built from the same small set of moves. Once you own the moves, you can invent the structure the problem actually calls for, rather than memorising a catalogue of well-known ones. Nevertheless, if you're serious about programming it's worth learning about the named structures. There are plenty of books on the topic, search for "Data Structures & Algorithms"; this chapter will give you a solid foundation for further study.

11.1 What Is a Pointer?

A variable normally *contains* a value: an integer, a real number, a character. A **pointer** is different: it contains the *address* of another variable in memory — it tells you *where* another value lives, rather than giving you the value itself.

Think of it this way. Suppose you want to find a book in a library. One approach is to carry the book with you — that is like a regular variable, which carries its value directly. Another approach is to carry a piece of paper with the book's shelf location written on it — that is like a pointer. The paper does not contain the book; it tells you where to find it.

This **indirection** is the essence of a pointer. A pointer variable holds a memory address. By **dereferencing** the pointer — following that address — you reach the actual data.



`p` itself contains an address. Following the arrow gives you the value at that address. Everything in this chapter is a variation on that one picture: a small box holding an address, and an arrow showing where it leads.

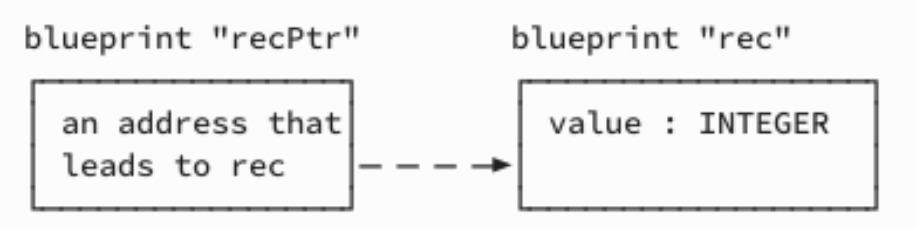
11.2 Pointer Types in Oberon-07

In Oberon-07, pointers always point to **records**. You cannot have a pointer to a plain integer or array — only to a record type. This keeps the type system safe: whenever you dereference a pointer, the compiler already knows exactly what shape of data is on the other end of the arrow.

A pointer type is declared as `POINTER TO` followed by the record type it points to:

```
TYPE
  recPtr = POINTER TO rec;
  rec    = RECORD
    value : INTEGER
  END;
```

This is a *blueprint*, not yet any actual data in memory. No record exists yet — we have only described what one would look like:



Notice that `recPtr` is declared *before* `rec`, even though `rec` has not been defined yet at that point. Oberon-07 allows this one exception to its usual "declare before use" rule — a pointer type may refer to a record type declared later in the same `TYPE` section. The compiler flags `rec` as a type it does not yet know about and comes back to it when it is eventually defined. As we will see in Section 11.7, a record type is often going to need to refer to *itself*, which is impossible unless this ordering is allowed.

A word on the name `Node` used above: it is only a name. Nothing about `POINTER TO Node` requires the record it points to to be part of a chain or any other connected structure (which is what the word "node" might suggest). A record becomes a "node" (in the sense of being part of a larger structure) only once you actually give it a pointer field and link it to something else, which we'll be doing later in this chapter.

Oberon-07 allows a pointer to lead only to a record — never directly to a plain `INTEGER`, `ARRAY`, or

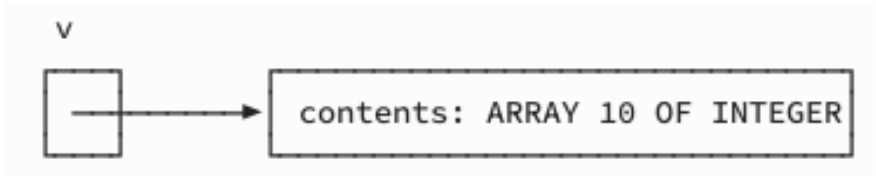
other non-record type. At first this looks like a limitation: what if you want a pointer to an array, or to a single real number?

The restriction turns out not to cost you anything, because a record is allowed to contain exactly one field of whatever type you need. Wrapping that type in a one-field record and pointing to the record achieves the same effect:

```
TYPE
  RealPtr  = POINTER TO RealBox;
  RealBox  = RECORD
    contents : REAL
  END;

  VecPtr   = POINTER TO VecBox;
  VecBox   = RECORD
    contents : ARRAY 10 OF INTEGER
  END;

VAR v : VecPtr;
```



`v^.contents` behaves exactly like a dynamically-allocated array — you index it as `v^.contents[i]`, and it exists independently for as long as something points to it. The record is doing nothing more than giving the pointer something to legally point *to*; the "real" data of interest is the single field inside. We will see in chapters 13 and 14 that such records can be useful.

11.3 NIL: The Pointer to Nothing

Before a pointer variable is given an address, it needs a well-defined starting value. Oberon-07 provides the special value `NIL`, meaning "this pointer leads nowhere":

```
VAR
  p : recPtr;
...
p := NIL;
```

Always initialise pointer variables to `NIL` before use. An uninitialised pointer has an undefined value — it might *look* like an address, but following it leads to random memory, and the program will trap. `NIL` is the only pointer value that is safe to hold without pointing at a real record, and it is the value you test for before ever writing `p^` — a pattern we will use constantly:

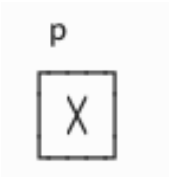
```
IF p = NIL THEN
  Out.String("p points to nothing")
END
```

11.4 Allocating a Record: NEW

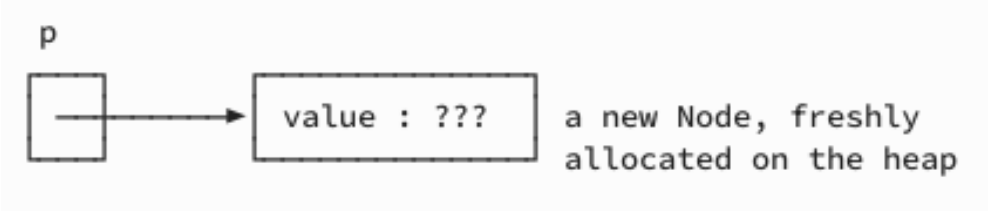
To bring a record into existence in dynamic memory, use the predefined procedure `NEW`:

```
NEW(p);
```

Before:



After:

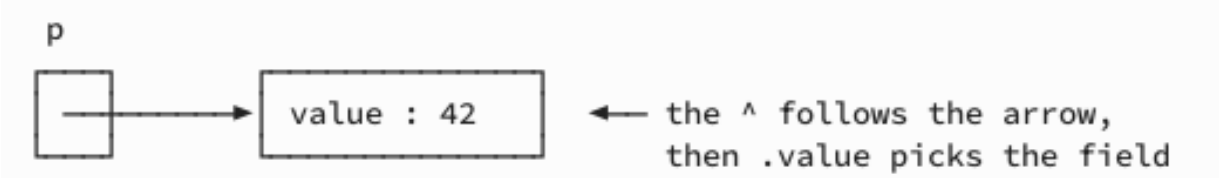


`NEW(p)` does two things in one step: it finds a free block of memory large enough for a record (in an area called the **heap**), and it stores that block's address in `p`. After this call, `p` is no longer `NIL`. The record's field has not been given a meaningful value yet — that is the next operation.

11.5 The Dereference Operator ^

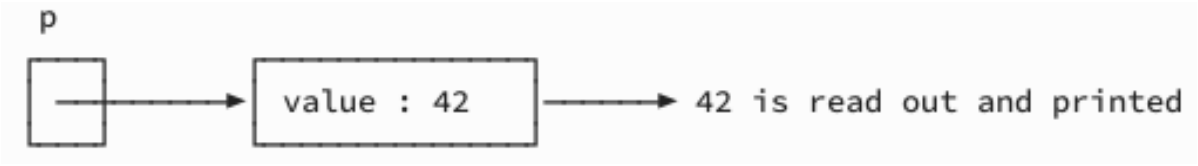
To reach the record a pointer points to, you **dereference** it with the `^` operator. `p` is the address; `p^` is the record living at that address; `p^.value` is the `value` field inside that record.

```
p^.value := 42;
```



The ^ makes the indirection explicit: you are following a pointer to reach a record, not accessing a field of a plain record variable sitting directly in p. Reading works the same way, following the same arrow:

```
Out.Int(p^.value, 0);
```



Oberon-07 also allows the shorthand p.value, letting the compiler insert the dereference automatically. In this chapter we will always write p^.value, so the indirection stays visible while the mental model is still new to you.

A complete program using everything so far:

```
MODULE PointerDemo;
  IMPORT Out;

TYPE
  recPtr = POINTER TO rec;
  rec    = RECORD
    value : INTEGER
  END;

VAR
  p, q : recPtr;

BEGIN
  p := NIL;              (* start with no record         *)
  NEW(p);                (* allocate one; p now points to it *)
  p^.value := 42;        (* write through the pointer     *)

  q := NIL;
  NEW(q);                (* a second, independent record  *)
  q^.value := 99;

  Out.String("p^.value = "); Out.Int(p^.value, 0); Out.Ln;
  Out.String("q^.value = "); Out.Int(q^.value, 0); Out.Ln
END PointerDemo.
```

Output:

```
p^.value = 42
q^.value = 99
```

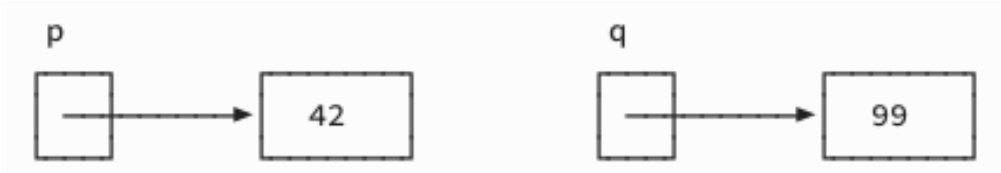
Two calls to `NEW` created two separate records. `p` and `q` each hold the address of their own record — the two boxes on the right are entirely independent.

11.6 Pointer Assignment: Sharing, Not Copying

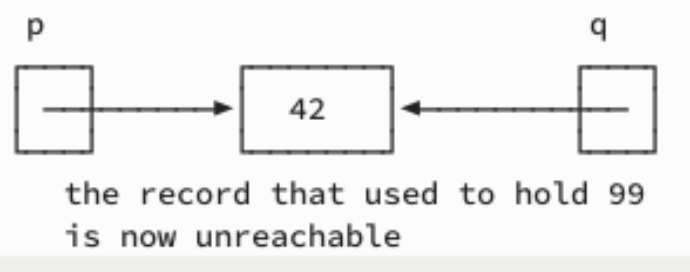
Pointers can be assigned to one another. This is where pointers start to behave differently from every other variable we have used.

```
q := p;
```

Before (continuing from `p^.value = 42`, `q` pointing at its own record containing 99):

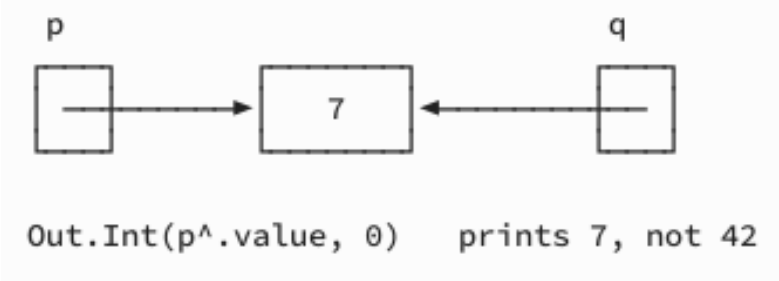


After `q := p`:



`q := p` did not copy the record — it copied the *address*. `p` and `q` now both lead to the same box. Changing the record through one pointer changes what the other one sees, because there is only one record:

```
q^.value := 7;
```



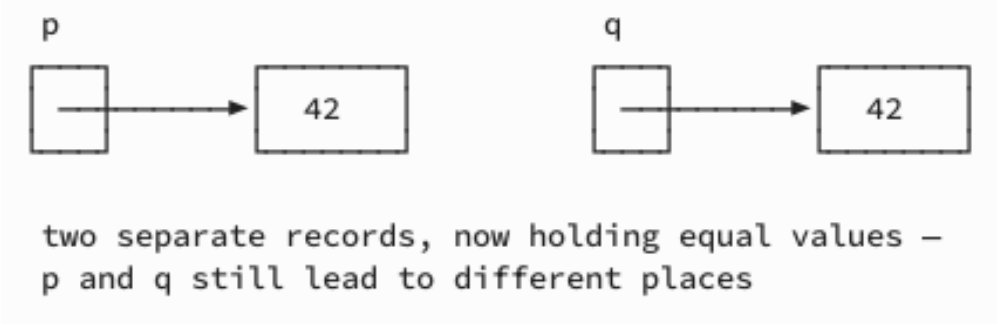
Compare this carefully with record variables from Chapter 10, where `r2 := r1` copies every field and the two variables are afterwards completely independent. Pointer assignment is the opposite: it makes two variables refer to *one shared* piece of data. Neither behaviour is "the correct one" — they are different tools, and confusing them is one of the most common sources of pointer bugs. Whenever you see `q := p` for pointers, read it as "`q` now leads to the same place `p` does," never as "`q` now equals a copy of what `p` has."

There is one further variation worth seeing now, because it is easy to confuse with `q := p`, and the `^` is what tells them apart. Suppose instead you write:

```
q^ := p^;
```

Before, with `p` and `q` pointing to two separate records:

After `q^ := p^`:



`q := p` (no carets) copies an *address*, leaving one record shared by two pointers. `q^ := p^` (both sides dereferenced) copies the *record itself* — every field of `p^` is copied into the corresponding field of `q^` — and `p` and `q` remain as separate as they were before, each still leading to its own record. The `^` is doing real work here: without it, `q := p` would silently mean something completely different from `q^ := p^`. This is exactly the field-by-field copy record variables get automatically (Chapter 10); the caret is what lets you ask for that same behaviour through a pointer instead of getting a shared reference by default.

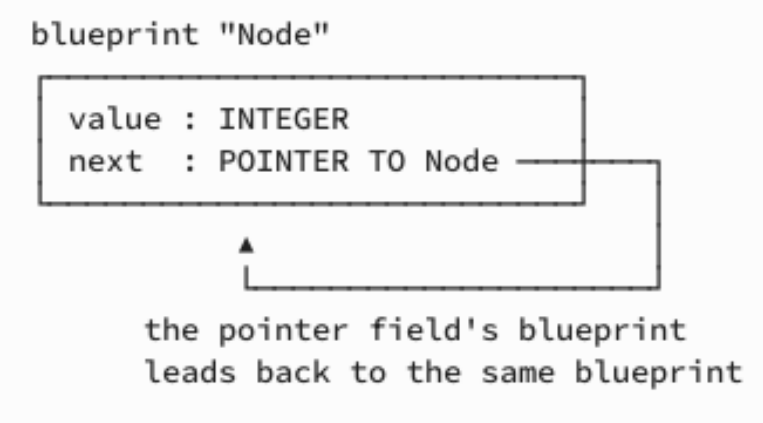
11.7 Self-Referential Records: The Key to Arbitrary Structures

Everything above lets you allocate independent, isolated records. To build a *structure* — many records connected to one another in some deliberate shape — a record needs to be able to point to another record of its own type. This is called a **self-referential record**, and it is the single idea that unlocks every dynamic structure:

```
TYPE
  NodePtr = POINTER TO Node;
  Node    = RECORD
```

```
    value : INTEGER;
    next  : NodePtr      (* a Node can point to another Node *)
  END;
```

It is common to call a record of this form a **node**.



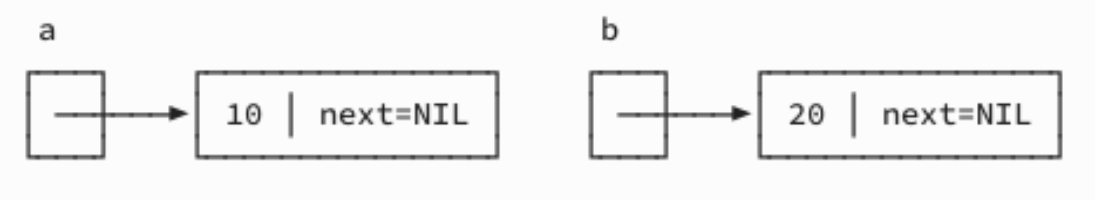
Nothing here has happened yet at runtime — this is still just a shape described in `TYPE`. What it makes *possible* is that once you allocate several `Node` records, each one has a spare pointer field capable of leading to another one of them. How you use that field — to build a straight chain, a branching structure, a ring, or something with no established name — is entirely up to you, and is the subject of the rest of this chapter. The field does not have to be called `next`; that is just a name we choose because it reads well for a chain. Section 11.11 uses different names for the same idea.

11.8 Linking Two Nodes Together

This is the one fundamental operation that every dynamic structure in this book is built from: **take two records that already exist, and store one's address in a pointer field of the other.**

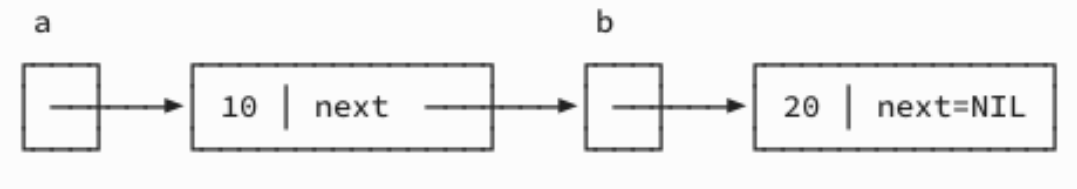
Start with two separate, unconnected nodes:

```
NEW(a); a^.value := 10; a^.next := NIL;
NEW(b); b^.value := 20; b^.next := NIL;
```



Now link them:

```
a^.next := b;
```

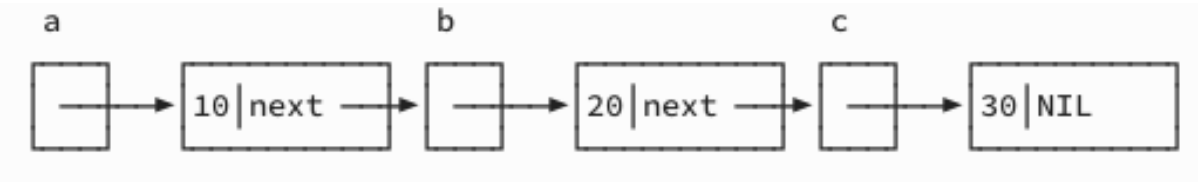


That single assignment is the whole operation. It looks identical to any other pointer assignment (Section 11.6) — the only difference is that this time the value being stored is the address of another node of the *same type*, so the two records are now connected. "Appending a node," "adding a child," "attaching a neighbour" — every such phrase boils down to this one statement: `somePointerField := someOtherNode`.

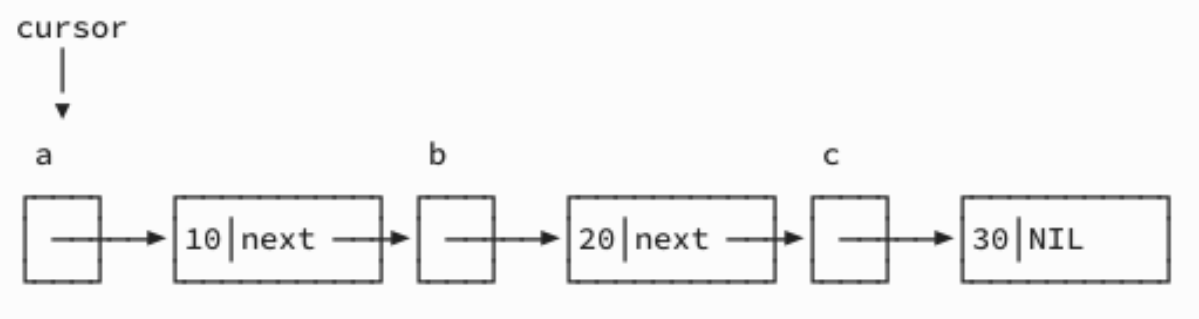
Linking says nothing about *order of creation*. `b` could have been allocated first, or `a` and `b` could be far apart in the code. What matters for the shape of the structure is only which pointer fields were made to lead where — a point Exercise 11.4 will ask you to rely on directly.

11.9 Traversing a Chain of Pointers

Once nodes are linked, you need a way to visit them. A **cursor** is an ordinary pointer variable used as a temporary "current position," which you advance one link at a time. Given the chain `a -> b -> c` built by linking as in the previous section:

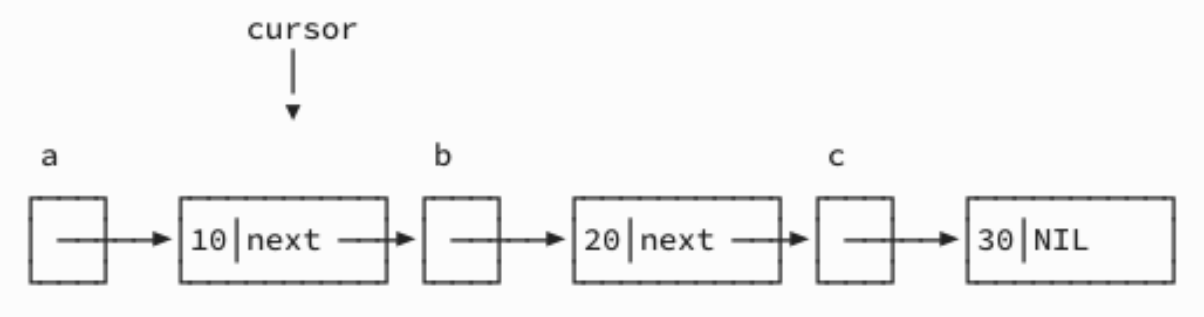



```
cursor := a;
```




```
cursor := cursor^.next;
```

Repeating `cursor := cursor^.next` walks `cursor` one link further each time, until it reaches a record whose `next` field is `NIL` — or becomes `NIL` itself, if the chain ends there. Wrapped in a loop:



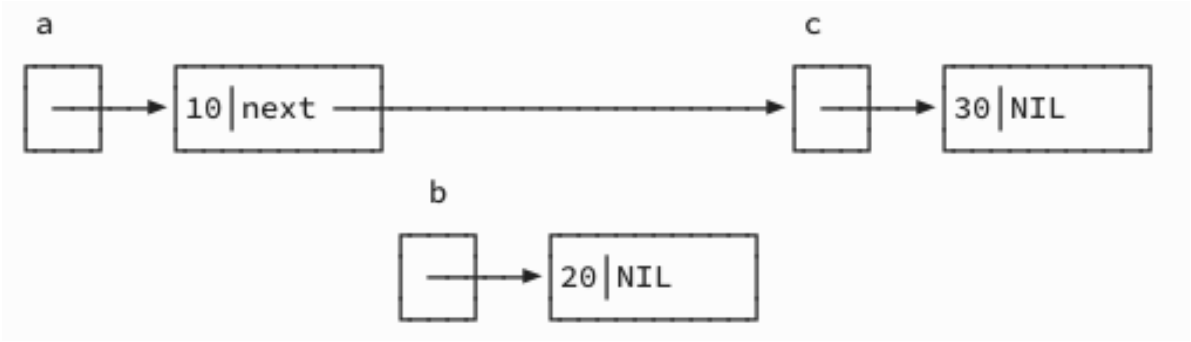

```
cursor := a;
WHILE cursor # NIL DO
  Out.Int(cursor^.value, 0); Out.String(" ");
  cursor := cursor^.next
END;
Out.Ln
```

The loop visits every record reachable by repeatedly following one particular pointer field, starting from a given node — a description that fits a linked list, a single branch of a tree, or any other chain you build with the linking operation from 11.8. The `#NIL` test is what makes the loop stop instead of running forever or trapping.

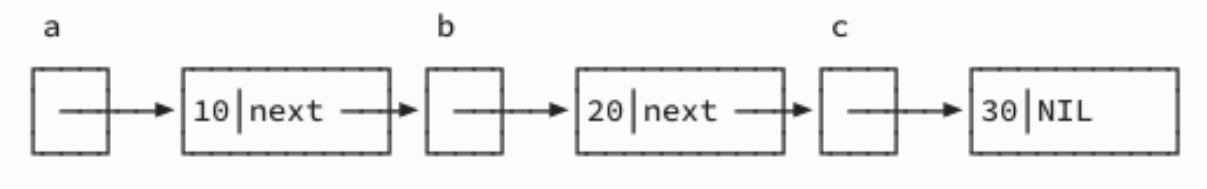
11.10 Redirecting a Pointer

The last fundamental operation is **redirection**: changing what an *existing* pointer field points to, once a structure is already built. This single move, used in two different ways, is what "inserting" and "removing" a node turn out to mean.

Splicing a node in between two existing nodes. Start with `a -> c` already linked, and a new node `b` allocated but not yet connected:

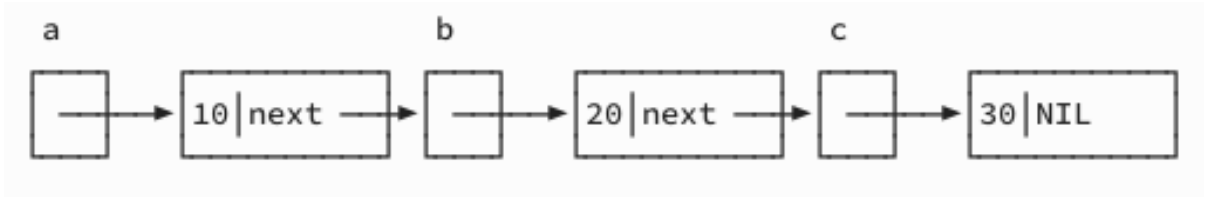



```
b^.next := a^.next;    (* b now points to what a used to point to *)
a^.next := b;          (* a is redirected to point to b instead   *)
```

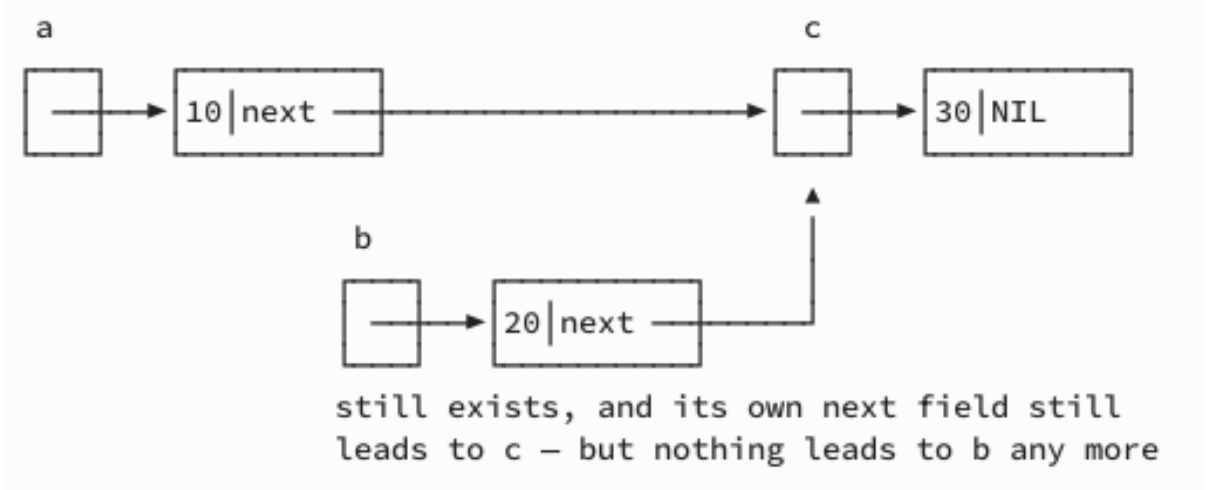


The order of those two lines matters: `b^.next` must be set to `a^.next` *before* `a^.next` is overwritten, or the address of `c` would be lost before `b` had a chance to record it.

Skipping over a node to remove it. Start with `a -> b -> c` linked, and suppose `b` should be removed:




```
a^.next := b^.next;   (* a is redirected past b, straight to c *)
```



`b`'s own record is untouched — its `next` field still points to `c`, exactly as before. What changed is that nothing leads *to* `b` any more. In OBNC the garbage collector will reclaim memory that has become unreachable like this; you are not required to free it manually, as in some languages like C.

Notice that both cases used the exact same kind of statement, `somePointerField := someAddress`. Insertion and removal are not two different mechanisms to memorise — they are the redirection operation from this section, aimed at two different goals.

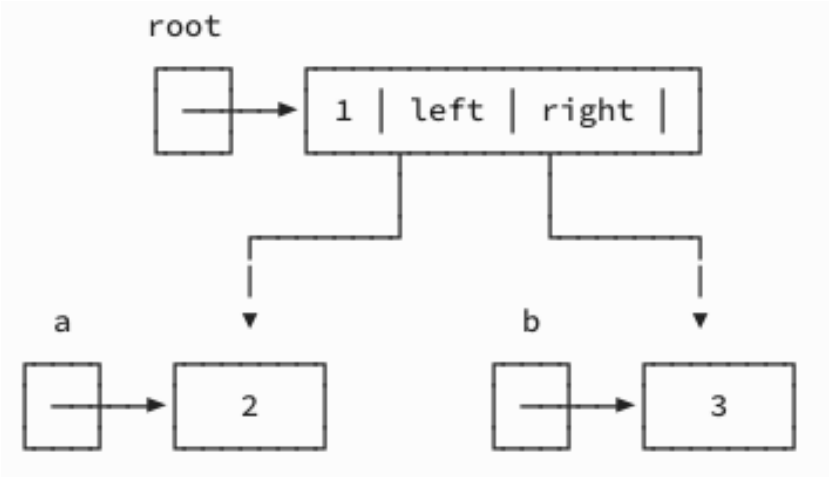
11.11 More Than One Pointer Field

Nothing about Sections 11.7–11.10 required a node to have only *one* pointer field. A record can declare several, each independently pointing (or not) to another node — and the same allocate / dereference / link / traverse / redirect operations apply to every one of them individually.

```
TYPE
  BranchPtr = POINTER TO Branch;
  Branch    = RECORD
    value    : INTEGER;
    left, right : BranchPtr
  END;

NEW(root);  root^.value := 1;  root^.left := NIL; root^.right := NIL;
NEW(a);     a^.value    := 2;  a^.left    := NIL; a^.right    := NIL;
NEW(b);     b^.value    := 3;  b^.left    := NIL; b^.right    := NIL;

root^.left  := a;      (* link root's left field to a  *)
root^.right := b;       (* link root's right field to b *)
```



The `left` field of `root` leads to `a`; the `right` field of `root` leads to `b`.

Two `link` statements, aimed at two different fields of the same node, produced a branch instead of a chain. Traversal still works exactly as in Section 11.9, except that reaching a node now offers a *choice* of which field to follow next — `cursor^.left` or `cursor^.right` — rather than a single, forced `cursor^.next`. Redirection still works exactly as in Section 11.10, field by field.

This is the point of separating the operations from the structures: `Node` with a `next` field, `Branch` with `left`/`right` fields, or a record you design yourself with three pointer fields named however you like, are all built and manipulated with the same five moves. The *shape* that results is entirely a consequence of which fields you choose to link to which nodes — a decision you now have full control over.

11.12 Example: Putting the Fundamentals Together

The following program uses every operation from this chapter in one place: allocation, dereferencing, linking three nodes into a chain, traversing it, redirecting a link to insert a node, and redirecting again to remove one.

```
MODULE PointerOps;
```

```
  IMPORT Out;

TYPE
  NodePtr = POINTER TO Node;
  Node    = RECORD
    value : INTEGER;
    next  : NodePtr
  END;

VAR
  a, b, c, cursor : NodePtr;

BEGIN
  (* --- allocate three independent nodes --- *)
  NEW(a); a^.value := 10; a^.next := NIL;
  NEW(b); b^.value := 30; b^.next := NIL;
  NEW(c); c^.value := 20; c^.next := NIL;

  (* --- link a to b, forming the chain 10 -> 30 --- *)
  a^.next := b;

  (* --- traverse and print --- *)
  Out.String("Chain: ");
  cursor := a;
  WHILE cursor # NIL DO
    Out.Int(cursor^.value, 0);
    IF cursor^.next # NIL THEN Out.String(" -> ") END;
    cursor := cursor^.next
  END;
  Out.Ln;

  (* --- redirect: splice c (20) between a and b --- *)
  c^.next := a^.next;
  a^.next := c;

  Out.String("After inserting 20: ");
  cursor := a;
  WHILE cursor # NIL DO
    Out.Int(cursor^.value, 0);
    IF cursor^.next # NIL THEN Out.String(" -> ") END;
    cursor := cursor^.next
  END;
  Out.Ln;

  (* --- redirect: skip over c (20), removing it --- *)
  a^.next := c^.next;
```

```
  Out.String("After removing 20: ");
  cursor := a;
  WHILE cursor # NIL DO
    Out.Int(cursor^.value, 0);
    IF cursor^.next # NIL THEN Out.String(" -> ") END;
    cursor := cursor^.next
  END;
  Out.Ln
END PointerOps.
```

Output:

```
Chain: 10 -> 30
After inserting 20: 10 -> 20 -> 30
After removing 20: 10 -> 30
```

Every line in this program is one of the five operations you have now seen diagrammed on its own: `NEW` allocates, `^` reads and writes, a plain assignment between two pointer fields links or redirects, and a `WHILE` loop with a cursor traverses. Any data structure - whether it be a linked-list, tree queue, stack or non-standard structure you invent consists of only these same five operations, applied with a plan.

A Checklist for Working with Pointers

Always initialise pointer variables to `NIL`. An uninitialised pointer has an undefined value; following it will cause a runtime trap.

Always check for `NIL` before dereferencing a pointer. The pattern `IF p # NIL THEN ... p^.field ... END` is the safe idiom, and it is worth making automatic before you build anything larger.

In a `WHILE` loop with two conditions, put the `NIL` check first. Write `WHILE (p # NIL) & (p^.value # x) DO`, not the other way around. Short-circuit evaluation ensures `p^.value` is only accessed once `p # NIL` is already known to be true.

Remember that pointer assignment shares data, it does not copy it. Writing `q := p` makes `q` lead to the same record as `p`. Writing `b^.next := a^.next` copies an *address* out of one field into another — it does not touch the record that address leads to.

When redirecting a pointer to splice something in, capture the old destination first. `b^.next := a^.next` must happen before `a^.next` is overwritten, or the rest of the structure becomes unreachable.

A field pointing to `NIL` is not a special case to work around — it is the normal way a chain of pointers ends. Every traversal loop in this chapter relies on it.

Summary

- A **pointer** holds the address of a record. Reaching the record through a pointer is called **dereferencing**, written `p^`. Oberon-07 also permits `p.field` as shorthand, but `p^.field` keeps the indirection visible.
- Pointer types are declared as `POINTER TO RecordType`; in Oberon-07, pointers can only lead to records.
- `NEW(p)` allocates a fresh record on the heap and stores its address in `p`.
- `NIL` means "leads nowhere." Initialise pointers to it, and test for it before every dereference.
- **Pointer assignment** (`q := p`) copies an address, making two variables lead to the same record — it does not copy the record's contents.
- A **self-referential record** — one whose type contains a pointer to its own type — is what makes open-ended structures possible.
- **Linking** two nodes means storing one node's address in a pointer field of another. It is an ordinary assignment; what makes it "linking" is only that the value assigned is another node's address.
- **Traversing** means following one pointer field repeatedly with a cursor, from a starting node until a `NIL` is reached.
- **Redirecting** a pointer field — changing what it points to after the structure already exists — is the operation behind both insertion (splice in a new destination) and removal (skip past the old one).
- A record may have **more than one** pointer field. The same five operations apply to each field independently; which fields you choose to link produces a chain, a branch, or any other shape the problem needs.

The Five Fundamental Operations

Everything in this chapter reduces to five moves. Every dynamic structure you build for the rest of this book — in the exercises below, and in later chapters — will be some combination of these:

1. **Allocate** — `NEW(p)` brings a new record into existence on the heap and stores its address in `p`.
2. **Dereference** — `p^` reaches the record `p` points to, so that `p^.field` can be read or written.
3. **Link** — storing one node's address in a pointer field of another (`a^.next := b`), connecting two records that already exist.
4. **Traverse** — following a pointer field repeatedly with a cursor, from a starting node until `NIL` is reached (or, for a structure with no `NIL` to reach, until some other stopping condition is met).
5. **Redirect** — reassigning an existing pointer field to a new destination, which is the operation behind both inserting a node (splicing in a new destination) and removing one (skipping past the old one).

Exercises

Note: It will be helpful when tackling these exercises to draw diagrams of the structure required before any coding. Indeed, this should always be done when designing an algorithm which uses pointers to create data structures.

- **Draw the "before" state first**, with every pointer you'll touch clearly labelled — including ones like `prev` or `cursor` that are just bookkeeping, not part of the permanent structure.

- **Draw the "after" state you want**, then compare the two and ask: which arrows changed, and in what order must they change so that nothing gets lost along the way?
- **Draw edge cases separately**, not just the typical case. An empty structure, a one-node structure, and "the node being removed is the head/tail" almost always need their own quick sketch, because they're where the general-case code tends to break.
- For anything beyond a straight chain, sketch **before deciding on field names**. It's easy to bake in an assumption (like "there's always a `next`") that turns out to be wrong once you draw the actual shape you need.

Pointer bugs are almost always about getting the order of operations wrong (the classic being overwriting a pointer before you've saved what it used to point to, which you saw in the splice example in 11.10). A diagram forces you to see the "before" and "after" states explicitly, which makes that kind of ordering mistake visible before you've written a line of code.

11.1 Declare a `Node` type with an `INTEGER` field and a `next` field. Without using a loop, write code that allocates three nodes by hand, gives them the values 5, 15, and 25, and links them into a chain in that order using three separate assignment statements. Draw the resulting structure (on paper or in comments) before writing the traversal that prints it.

11.2 Allocate a single node and, instead of linking it to a different node, set its own `next` field to point to *itself*. Write a loop that traverses starting from this node, printing its value and a running count each time around, stopping the loop with a counter after 5 iterations rather than relying on `# NIL` (which will never become true). Explain in a comment why this structure can never be traversed to completion by a plain `WHILE cursor # NIL` loop.

11.3 Declare a record type with **three** pointer fields, `p1`, `p2`, and `p3` (any names you like), each `POINTER TO` the same record type. Allocate three nodes, `x`, `y`, and `z`, and link them into a triangle: `x^.p1` leads to `y`, `y^.p1` leads to `z`, and `z^.p1` leads back to `x`. Write a loop that starts at `x` and prints each node's value once per lap, for exactly 3 laps, using a counter to stop (again, `NIL` never appears).

11.4 Allocate four nodes, `n1`, `n2`, `n3`, `n4`, giving them the values 100, 200, 300, 400 — but link them **out of allocation order**, so that following `next` from `n1` visits 100, then 300, then 200, then 400. Confirm your chain is correct by traversing and printing it. The purpose of this exercise is to make the point of Section 11.8 concrete: the shape of a structure depends only on how pointer fields are linked, never on the order in which the underlying records were created.

11.5 Using the `Branch` type from Section 11.11 (a `value` field plus `left` and `right` pointer fields), build a small structure of five nodes : a root with two children, and one of those children with a child of its own. Write a traversal that visits the root, then everything reachable via `left`, then everything reachable via `right`.

11.6 Take the three-node chain built by the redirection example in Section 11.10 (`a -> b -> c`, values 10, 20, 30) and, using only the redirection operation, reverse the order of the links so that traversing from `c` now reaches `b` then `a` (you will need a field capable of leading backward — either add a second pointer field to `Node`, or store `a`, `b`, and `c` in separate variables and rewrite each `next` field explicitly). Print the chain in its new order to confirm.

11.7 Design your own record type with at least two pointer fields and at least one data field, and use it to build any structure of five or more nodes that is *not* a straight chain and *not* the triangle from Exercise 11.3 — the shape and its purpose are entirely up to you. In a comment, briefly describe what shape you built and which of the five fundamental operations (allocate, dereference, link, traverse,

redirect) you used to construct it.

Chapter 12 — Sets

The final data type we will cover in Part II is the **set**. Unlike arrays and records, which are structures that hold multiple named or indexed values, a set represents something more mathematical: a collection of distinct members, where the only question you can ask about any given value is whether or not it belongs to the collection.

Sets are not used as frequently as the other types in this book, but they are genuinely useful for certain problems — particularly when you need to track membership, test combinations of conditions, or manage groups of flags. Oberon-07's `SET` type provides a clean and efficient way to do all of these things.

12.1 What Is a Set?

In mathematics, a set is a collection of distinct elements with no particular order. The set {1, 3, 5} contains three elements; {3, 1, 5} is the same set. An element is either in the set or it is not — it cannot appear twice.

Oberon-07's `SET` type represents sets of small non-negative integers. Specifically, a `SET` value is a collection of integers chosen from the range 0 to a system-dependent maximum. On a 64-bit system using OBNC, that maximum is 63, so a `SET` can contain any subset of the integers 0 through 63.

This may seem restrictive, but it covers a wide range of practical uses. The integers in the set are often used as codes or indices rather than values in themselves — for example, a set of permission flags, a set of selected menu options, or a set of active states.

Internally, a `SET` is stored as a single machine word in which each bit position corresponds to one integer. Bit 0 corresponds to the element 0, bit 1 to element 1, and so on. This means set operations are extremely fast — they map directly onto the processor's bitwise operations.

12.2 Declaring and Assigning Sets

A `SET` variable is declared in the usual way:

```
VAR
  s : SET;
```

A set value is written as a comma-separated list of elements enclosed in curly braces:

```
s := {1, 3, 5, 7};     (* s contains 1, 3, 5, and 7 *)
s := {0};              (* s contains only 0 *)
s := {};               (* s is the empty set *)
```

Elements can also be specified as a **range** using `..`:

```
s := {1..5};           (* s contains 1, 2, 3, 4, 5 *)
s := {0..7};           (* s contains 0, 1, 2, 3, 4, 5, 6, 7 *)
```

Ranges and individual elements can be combined in a single literal:

```
s := {0, 2..5, 9};    (* s contains 0, 2, 3, 4, 5, and 9 *)
```

The elements in a set literal must be integers in the valid range (0 to 63 in OBNC). Using a value outside this range causes a runtime trap.

12.3 The IN Operator

The most fundamental set operation is testing membership: is a given integer an element of a given set? The `IN` operator performs this test and returns a `BOOLEAN` result:

```
x IN s
```

This is `TRUE` if the integer `x` is a member of the set `s`, and `FALSE` otherwise. For example:

```
s := {1, 3, 5, 7};
Out.Int(ORD(1 IN s), 0);    (* prints 1 -- TRUE *)
Out.Int(ORD(2 IN s), 0);    (* prints 0 -- FALSE *)
Out.Int(ORD(5 IN s), 0);    (* prints 1 -- TRUE *)
```

Note that the above code works because the expression `1 IN s` returns a `BOOLEAN` value and `ORD` is defined for `BOOLEAN` as well as for `CHAR`and `SET`: `ORD(TRUE)` returns 1 and `ORD(FALSE)` returns 0.

`IN` is most commonly used in `IF` statements:

```
IF day IN {6, 7} THEN
  Out.String("Weekend")
ELSE
  Out.String("Weekday")
END
```

This is cleaner and more readable than the equivalent using `OR`:

```
IF (day = 6) OR (day = 7) THEN ...
```

The `IN` form scales gracefully: testing whether a value belongs to a set of ten elements is just as concise as testing for two.

Example 12.1: Vowel Testing with IN

In Chapter 9 we tested for vowels using a long chain of `OR` comparisons. Sets offer a cleaner alternative. Because `CHAR` values can be converted to their ordinal integers with `ORD`, we can build a set of vowel ordinals and test membership directly. Note that since the ordinal values for the lowercase vowels start at 97, which is larger than the maximum integer allowed for a set, we can use the simple trick of subtracting a suitable value from each ordinal value (namely, 60) and likewise

subtracting this value from `ORD(ch)` in the `IF` statement which tests for a vowel:

```
MODULE VowelSet;
  IMPORT In, Out;
VAR
  ch     : CHAR;
  vowels : SET;
BEGIN
  (* ordinals of lowercase vowels: a=97, e=101, i=105, o=111, u=117 *)
  vowels := {37, 41, 45, 51, 57};  (* subtract 60 from the above values *)
  Out.String("Enter a lowercase letter: ");
  Out.Ln;
  In.Char(ch);
  IF ORD(ch)-60 IN vowels THEN
    Out.String("It is a vowel.")
  ELSE
    Out.String("It is not a vowel.")
  END;
  Out.Ln
END VowelSet.
```

12.4 Set Operators

Oberon-07 provides four operators for combining sets. They use the same symbols as arithmetic operators, but when applied to `SET` operands they perform set operations rather than arithmetic.

Union: `+`

The union of two sets contains every element that is in either set (or both):

```
{1, 2, 3} + {3, 4, 5}    (* result: {1, 2, 3, 4, 5} *)
```

Difference: `-`

The difference `a - b` contains every element that is in `a` but not in `b`:

```
{1, 2, 3, 4} - {2, 4}    (* result: {1, 3} *)
```

The unary minus applied to a single set gives the **complement** — every element from 0 to 63 that is *not* in the set:

```
- {0, 1, 2}    (* result: {3, 4, ..., 63} *)
```

Intersection: *

The intersection of two sets contains only the elements that are in both sets:

```
{1, 2, 3} * {2, 3, 4}    (* result: {2, 3} *)
```

Symmetric Difference: /

The symmetric difference contains elements that are in one set or the other, but not in both — the opposite of intersection:

```
{1, 2, 3} / {2, 3, 4}    (* result: {1, 4} *)
```

Equality and Inequality

Two sets can be compared for equality with = and #:

```
{1, 2, 3} = {3, 1, 2}    (* TRUE: order does not matter *)
{1, 2}    # {1, 2, 3}    (* TRUE: different sets *)
```

Example 12.2: Set Operations

Here's a complete program which demonstrates the set operations.

```
MODULE SetOps;
  IMPORT Out;
VAR
  a, b, result : SET;
BEGIN
  a := {1, 2, 3, 4, 5};
  b := {3, 4, 5, 6, 7};
  result := a + b;
  Out.String("Union:               ");
  (* print members 0..9 *)
  Out.String("{");
  IF 1 IN result THEN Out.String("1 ") END;
  IF 2 IN result THEN Out.String("2 ") END;
  IF 3 IN result THEN Out.String("3 ") END;
  IF 4 IN result THEN Out.String("4 ") END;
  IF 5 IN result THEN Out.String("5 ") END;
  IF 6 IN result THEN Out.String("6 ") END;
  IF 7 IN result THEN Out.String("7 ") END;
  Out.String("}"); Out.Ln;
  result := a * b;
```

```
  Out.String("Intersection:        {");
  IF 3 IN result THEN Out.String("3 ") END;
  IF 4 IN result THEN Out.String("4 ") END;
  IF 5 IN result THEN Out.String("5 ") END;
  Out.String("}"); Out.Ln;
  result := a - b;
  Out.String("a minus b:           {");
  IF 1 IN result THEN Out.String("1 ") END;
  IF 2 IN result THEN Out.String("2 ") END;
  Out.String("}"); Out.Ln;
  result := a / b;
  Out.String("Symmetric difference:{");
  IF 1 IN result THEN Out.String("1 ") END;
  IF 2 IN result THEN Out.String("2 ") END;
  IF 6 IN result THEN Out.String("6 ") END;
  IF 7 IN result THEN Out.String("7 ") END;
  Out.String("}"); Out.Ln
END SetOps.
```

Output:

```
Union:               {1 2 3 4 5 6 7 }
Intersection:        {3 4 5 }
a minus b:           {1 2 }
Symmetric difference:{1 2 6 7 }
```

Printing a set is slightly awkward because Oberon-07 provides no built-in procedure for it. The standard approach, as shown above, is to test each possible member with `IN` and print it if present. For a general set-printing utility we would loop from 0 to 63 — we will see a cleaner version of this shortly.

12.5 INCL and EXCL

Two predefined procedures modify sets in place by adding or removing a single element.

`INCL(s, x)` includes integer `x` in set `s`. It is equivalent to `s := s + {x}`:

```
s := {1, 3, 5};
INCL(s, 7);    (* s is now {1, 3, 5, 7} *)
INCL(s, 3);    (* s is still {1, 3, 5, 7} — 3 was already present *)
```

`EXCL(s, x)` excludes integer `x` from set `s`. It is equivalent to `s := s - {x}`:

```
s := {1, 3, 5, 7};
EXCL(s, 3);    (* s is now {1, 5, 7} *)
```

```
EXCL(s, 9);    (* s is still {1, 5, 7} — 9 was not present *)
```

`INCL` and `EXCL` are more efficient than the equivalent assignment expressions because they modify the set in place without creating an intermediate value. They are the idiomatic Oberon way to update a set one element at a time.

12.6 Printing a Set

Because there is no built-in procedure for printing a `SET` value, it is worth writing a small loop that does so. The natural approach is to check every possible element in turn and print those that are present:

```
Out.String("{");
first := TRUE;
FOR i := 0 TO 63 DO
  IF i IN s THEN
    IF ~first THEN Out.String(", ") END;
    Out.Int(i, 0);
    first := FALSE
  END
END;
Out.String("}")
```

The `first` flag suppresses the comma before the first printed element, so the output looks like `{1, 3, 5, 7}` rather than `{, 1, 3, 5, 7}`. In practice you rarely need to print all 64 possible elements — if you know your set only uses elements in a small range you can adjust the loop bounds accordingly.

12.7 Sets as Flag Collections

One of the most practical uses of sets is representing a collection of **flags** — on/off switches that can be independently set or cleared. This is common in systems programming, but it arises in application programming too: a set of active features, a set of selected items, a set of error conditions.

The standard pattern is to define named constants for each flag, using small integers:

```
CONST
  CanRead    = 0;
  CanWrite   = 1;
  CanExecute = 2;
  IsHidden   = 3;
  IsSystem   = 4;
```

Then a `SET` variable holds whichever flags are currently active:

```
VAR
  permissions : SET;
```

```
...
permissions := {CanRead, CanWrite};    (* can read and write, not execute *)
```

Testing and modifying individual flags is then clear and readable:

```
IF CanRead IN permissions THEN
  Out.String("Read access granted.")
END;

INCL(permissions, CanExecute);        (* grant execute permission *)
EXCL(permissions, CanWrite);          (* revoke write permission  *)
```

This is far more readable than the equivalent using integers and bitwise arithmetic, and it is exactly the kind of problem sets were designed for.

12.8 A Simple Permission System

This program manages a small set of file permissions. The user can view the current permissions, grant or revoke individual permissions, and check whether a specific operation is allowed.

Example 12.3

```
MODULE Permissions;
  IMPORT In, Out;
CONST
  CanRead    = 0;
  CanWrite   = 1;
  CanExecute = 2;
VAR
  perms  : SET;
  choice : INTEGER;
BEGIN
  perms := {CanRead};    (* start with read permission only *)
  REPEAT
    Out.Ln;
    Out.String("Current permissions: ");
    IF CanRead    IN perms THEN Out.String("Read ")    END;
    IF CanWrite   IN perms THEN Out.String("Write ")   END;
    IF CanExecute IN perms THEN Out.String("Execute ") END;
    IF perms = {} THEN Out.String("(none)") END;
    Out.Ln;
    Out.Ln;
    Out.String("1. Grant Read");    Out.Ln;
    Out.String("2. Grant Write");   Out.Ln;
    Out.String("3. Grant Execute"); Out.Ln;
```

```
    Out.String("4. Revoke Read");   Out.Ln;
    Out.String("5. Revoke Write");  Out.Ln;
    Out.String("6. Revoke Execute");Out.Ln;
    Out.String("7. Check access");  Out.Ln;
    Out.String("0. Quit");          Out.Ln;
    Out.String("Choice: ");
    In.Int(choice);
    Out.Ln;
    IF In.Done & (choice IN {1..7}) THEN
      CASE choice OF
          1 : INCL(perms, CanRead)
        | 2 : INCL(perms, CanWrite)
        | 3 : INCL(perms, CanExecute)
        | 4 : EXCL(perms, CanRead)
        | 5 : EXCL(perms, CanWrite)
        | 6 : EXCL(perms, CanExecute)
        | 7 : Out.String("Read?    ");
        IF CanRead IN perms THEN
          Out.String("Yes")
        ELSE
          Out.String("No")
        END;
          Out.Ln; Out.String("Write?   ");
        IF CanWrite IN perms THEN
          Out.String("Yes")
        ELSE
          Out.String("No")
        END;
          Out.Ln; Out.String("Execute? ");
        IF CanExecute IN perms THEN
          Out.String("Yes")
        ELSE
          Out.String("No")
        END;
        Out.Ln
      END
    END
  UNTIL choice = 0
END Permissions.
```

A sample session:

```
Current permissions: Read

1. Grant Read
```

```
2. Grant Write
3. Grant Execute
4. Revoke Read
5. Revoke Write
6. Revoke Execute
7. Check access
0. Quit
Choice: 2

Current permissions: Read Write

...
Choice: 7

Read?    Yes
Write?   Yes
Execute? No
```

The `SET` operations keep the permission logic concise and readable. Each `INCL` or `EXCL` call does exactly what its name says, and each `IN` test asks a clear question. Compare this to the equivalent using integer flags and bitwise arithmetic — the set version is noticeably easier to read and to reason about.

12.9 Arrays of Sets

Arrays can contain any type, including sets. The following example simulates roulette, which is a popular casino game in which players bet on where a ball will land on a spinning wheel divided into numbered pockets. Bets can be placed on specific numbers, groups of numbers, colours (red or black), or whether the result will be odd/even or high/low. After betting closes, the wheel is spun and a small ball is released; the winning outcome is the pocket where the ball comes to rest. Payouts vary depending on the type of bet, with higher-risk bets offering larger rewards.

This program simulates four players sitting at a roulette table; an array of sets is used to hold their bets. It demonstrates how easily sets can handle different types of bets—from single numbers to entire blocks like “dozens”—and how cleanly we can check for winners.

Example 12.4

```
MODULE Roulette;
IMPORT Out, Rand;
CONST
  MaxPlayers = 4;
VAR
  (* Each index is a player. The SET holds the numbers they bet on. *)
  bets: ARRAY MaxPlayers OF SET;
  winningNumber: INTEGER;
```

```
  i: INTEGER;
BEGIN
  (* Initialize the table (clear all previous bets) *)
  FOR i := 0 TO MaxPlayers - 1 DO
    bets[i] := {}
  END;
  (* Players place their bets *)
  (* Player 0 bets on the 1st Dozen *)
  bets[0] := {1 .. 12};
  (* Player 1 bets on a few specific "lucky" numbers *)
  bets[1] := {7, 17, 23, 36};
  (* Player 2 bets on the 3rd Dozen AND the zero as an insurance policy *)
  bets[2] := {0, 25 .. 36};
  (* Player 3 adds to their bet dynamically using INCL *)
  bets[3] := {2, 4};        (* Starts with a couple of evens *)
  INCL(bets[3], 6);         (* Adds another number before the wheel spins *)
  (* The dealer spins the wheel... *)
  winningNumber := Rand.Range(37);
  Out.String("The winning number is: ");
  Out.Int(winningNumber, 0);
  Out.Ln;
  Out.Ln;
  (* The dealer checks all players' bets to see who won *)
  FOR i := 0 TO MaxPlayers - 1 DO
    Out.String("Player "); Out.Int(i, 0);
    (* The IN operator checks if the winning number is inside the player's set *)
    IF winningNumber IN bets[i] THEN
      Out.String(" WINS!")
    ELSE
      Out.String(" loses.")
    END;
    Out.Ln
  END
END Roulette.
```

sample run:

```
The winning number is: 32

Player 0 loses.
Player 1 loses.
Player 2 WINS!
Player 3 loses.
```

Summary

- The `SET` type represents a set of small non-negative integers. In OBNC on a 64-bit system, the valid elements are integers from 0 to 63.
- Set literals are written in curly braces: `{1, 3, 5}`. A range is written with `..`: `{1..5}`. The empty set is `{}`.
- The `IN` operator tests membership: `x IN s` is `TRUE` if integer `x` is an element of set `s`.
- Set operators: `+` (union), `-` (difference or complement), `*` (intersection), `/` (symmetric difference).
- Sets can be compared for equality (`=`) and inequality (`#`).
- `INCL(s, x)` adds element `x` to set `s`. `EXCL(s, x)` removes it. Both modify `s` in place.
- Sets have no built-in print procedure; print them by testing each element with `IN` in a loop.
- Sets are ideal for representing collections of **flags** or **permissions**: define named constants for the flag values and use `INCL`, `EXCL`, and `IN` to manage and test them.

Exercises

12.1 Write a program that declares a set containing the prime numbers less than 20 (`{2, 3, 5, 7, 11, 13, 17, 19}`), reads an integer from the user, and reports whether it is in the set.

12.2 Write a program that reads two integers `a` and `b` from the user (both in the range 0–63) and prints all integers from 0 to 63 that are in the range `{a..b}`. Use `IN` to test membership.

12.3 Write a program that builds two sets by reading integers from the user (one set at a time, terminated by -1) and then prints the union, intersection, and difference of the two sets.

12.4 Using the flag pattern from Section 12.7, write a program that tracks which days of the week (1 = Monday through 7 = Sunday) a user has marked as busy. The user can toggle any day on or off, and the program prints the current set of busy days after each change.

12.5 Write a program that reads a sentence from the user and reports which vowels (`a`, `e`, `i`, `o`, `u`) appear in it, using a `SET` to track which vowels have been seen. Print only the vowels that actually appear.

12.6 Write a program that uses a `SET` to find all integers from 2 to 50 that are prime, using the Sieve of Eratosthenes algorithm. Start with the set `{2..50}`. For each integer `i` from 2 to 7 (the square root of 50, rounded down), remove all multiples of `i` greater than `i` itself from the set. What remains is the set of primes. Print the result. (Hint: use a `FOR` loop with a `BY` clause to step through the multiples, and `EXCL` to remove each one.)

12.7 Extend the `Permissions` program from Section 12.8 to add a fourth permission, `IsReadOnly` (constant value 3), and update the menu and display accordingly. Add a rule: if `IsReadOnly` is set, the `CanWrite` permission should automatically be revoked when the user grants `IsReadOnly`.

Part III — Structuring Programs

- Chapter 13 — Procedures
- Chapter 14 — Function Procedures
- Chapter 15 — Recursion
- Chapter 16 — Modules

Chapter 13 — Procedures

Every program we have written so far has been a single, unbroken sequence of statements in a module's `BEGIN...END` block. This works for small programs, but as programs grow it becomes a problem. A long sequence of statements is hard to read, hard to understand, and hard to maintain. If the same pattern of statements appears in several places — printing a record, reading a value, performing a calculation — you have to write it out every time, and if you want to change it you have to find and update every copy.

The solution is to break programs into named, reusable pieces. In Oberon-07, these pieces are called **procedures**. A procedure is a named block of code that can be called — activated — from anywhere in the program. You write it once, give it a name, and invoke it whenever you need it. This chapter introduces procedures, explains how data flows in and out of them through parameters, and shows how they transform programs from monolithic sequences into clear, structured designs.

13.1 What Is a Procedure?

A procedure is a named group of declarations and statements that performs a specific task. When you call a procedure, execution jumps to that group of statements, carries them out, and then returns to wherever the call was made.

You have been using procedures throughout this book — `Out.String`, `In.Int`, `Strings.Append` are all procedures. Now you will learn to write your own.

The simplest possible procedure takes no inputs and produces no output — it just performs a fixed action:

```
PROCEDURE PrintLine;
BEGIN
  Out.String("--------------------");
  Out.Ln
END PrintLine;
```

This declares a procedure named `PrintLine`. To call it, write its name:

```
PrintLine
```

Each call to `PrintLine` executes its two statements. You can call it as many times as you like.

13.2 Declaring Procedures

A procedure declaration has the following structure:

```
PROCEDURE name;
  (* local declarations, if any *)
BEGIN
  (* statements *)
```

```
END name;
```

The procedure name appears both after `PROCEDURE` and after the closing `END`. This repetition is required by the language — it makes the structure visually clear when procedures are long.

Procedure declarations appear in the **declaration sequence** of a module, after `VAR` declarations and before `BEGIN`. Like other declarations, they are not executed when they are encountered — they simply define the procedure so it can be called later.

Example 13.1

```
MODULE LineDemo;
  IMPORT Out;

PROCEDURE PrintLine;
BEGIN
  Out.String("--------------------");
  Out.Ln
END PrintLine;

PROCEDURE PrintHeading;
BEGIN
  PrintLine;
  Out.String("   Report");
  Out.Ln;
  PrintLine
END PrintHeading;

BEGIN
  PrintHeading;
  Out.String("   Item 1");
  Out.Ln;
  Out.String("   Item 2");
  Out.Ln;
  PrintLine
END LineDemo.
```

Output:

```
--------------------
   Report
--------------------
   Item 1
   Item 2
--------------------
```

Notice that `PrintHeading` calls `PrintLine` — procedures can call other procedures. The module's main `BEGIN...END` block calls both. The order of declaration matters: a procedure must be declared before it is called. Since `PrintLine` is called by `PrintHeading`, `PrintLine` must be declared first.

13.3 Parameters

A procedure that always does exactly the same thing is useful but limited. Most procedures need to work with different data each time they are called. This is where **parameters** come in.

A parameter is a named placeholder in the procedure declaration. When the procedure is called, the caller provides an **argument** — an actual value — for each parameter. The procedure then uses those values in its statements.

Here is a complete program with a procedure that prints a label and an integer value on one line:

Example 13.2

```
MODULE PrintDemo;
  IMPORT Out;

PROCEDURE PrintInt(label : ARRAY OF CHAR; value : INTEGER);
BEGIN
  Out.String(label);
  Out.Int(value, 0);
  Out.Ln
END PrintInt;

BEGIN
  PrintInt("Score: ", 87);
  PrintInt("Count: ", 42);
  PrintInt("Total: ", 129)
END PrintDemo.
```

Output:

```
Score: 87
Count: 42
Total: 129
```

The parameter list `(label : ARRAY OF CHAR; value : INTEGER)` declares two parameters: `label` of type `ARRAY OF CHAR` (notice that this declaration doesn't mention the *size* of the array - more on this in Section 13.8) and `value` of type `INTEGER`. Each call provides its own arguments, and the procedure uses those specific values. The parameters `label` and `value` exist only inside the procedure — they are created when the procedure is called and disappear when it returns.

Parameters of the Same Type

Several parameters of the same type can be grouped:

```
PROCEDURE PrintSum(a, b : INTEGER);
BEGIN
  Out.Int(a + b, 0);
  Out.Ln
END PrintSum;
```

Multiple groups are separated by semicolons:

```
PROCEDURE PrintRect(width, height : INTEGER; label : ARRAY OF CHAR);
```

13.4 Value Parameters and VAR Parameters

There are two fundamentally different kinds of parameter in Oberon-07, and understanding the difference is essential.

Value Parameters

A **value parameter** receives a copy of the argument. The procedure works with its own local copy, and any changes it makes to that copy have no effect on the original variable in the caller. Value parameters are the default — no keyword is needed.

Example 13.3

```
MODULE ValueDemo;
  IMPORT Out;
VAR
  x : INTEGER;

PROCEDURE Double(n : INTEGER);
BEGIN
  n := n * 2;           (* modifies the local copy only *)
  Out.Int(n, 0);
  Out.Ln
END Double;

BEGIN
  x := 5;
  Double(x);            (* prints 10 *)
  Out.Int(x, 0);        (* prints 5 -- x is unchanged *)
  Out.Ln
END ValueDemo.
```

Output:

```
10
5
```

The procedure receives the *value* 5, doubles its local copy to 10 and prints it, then returns. The variable `x` in the caller is untouched.

VAR Parameters

A **VAR parameter** is different: instead of receiving a copy, the procedure receives a reference to the original variable. Any change the procedure makes to a `VAR` parameter is a change to the original. The keyword `VAR` precedes the parameter name in the declaration:

Example 13.4

```
MODULE VarDemo;
  IMPORT Out;
VAR
  x : INTEGER;

PROCEDURE Double(VAR n : INTEGER);
BEGIN
  n := n * 2            (* modifies the original variable *)
END Double;

BEGIN
  x := 5;
  Double(x);            (* x is now 10 *)
  Out.Int(x, 0);        (* prints 10 *)
  Out.Ln
END VarDemo.
```

Output:

```
10
```

Now `x` really is doubled. The procedure has modified the caller's variable directly.

`VAR` parameters serve two purposes: they allow a procedure to return results through its parameters, and they avoid copying large data structures (such as records or arrays) when that would be expensive.

When to Use Each Kind

Use a **value parameter** when the procedure only needs to read the value — it should not, and does

not, change the caller's variable.

Use a **VAR parameter** when the procedure needs to modify the caller's variable, or when you are passing a large record that would be expensive to copy.

The `In` module procedures are a familiar example: `In.Int(n)` takes `n` as a `VAR` parameter because its whole purpose is to write a value into `n`.

Example 13.5: Swap

A classic use of `VAR` parameters is a swap procedure — exchanging the values of two variables. Without `VAR` parameters this would be impossible, since the procedure would only be working with copies:

```
MODULE SwapDemo;
  IMPORT Out;
VAR
  x, y : INTEGER;

PROCEDURE Swap(VAR a, b : INTEGER);
VAR
  temp : INTEGER;
BEGIN
  temp := a;
  a    := b;
  b    := temp
END Swap;

BEGIN
  x := 10;
  y := 20;
  Out.String("Before: ");
  Out.Int(x, 0); Out.String(" "); Out.Int(y, 0); Out.Ln;
  Swap(x, y);
  Out.String("After:  ");
  Out.Int(x, 0); Out.String(" "); Out.Int(y, 0); Out.Ln
END SwapDemo.
```

Output:

```
Before: 10 20
After:  20 10
```

Notice that the `temp` variable is declared inside `Swap`; it is a **local variable** — it exists only for the duration of the procedure call, as we discuss in the next section.

13.5 Local Variables

Just as a procedure can have its own parameters, it can have its own **local variables** — declared in a `VAR` section inside the procedure body. Local variables exist only while the procedure is executing. They are created fresh (with undefined values) each time the procedure is called and disappear when it returns.

```
PROCEDURE PrintFactors(n : INTEGER);
VAR
  i : INTEGER;
BEGIN
  Out.String("Factors of ");
  Out.Int(n, 0);
  Out.String(": ");
  FOR i := 1 TO n DO
    IF n MOD i = 0 THEN
      Out.Int(i, 0);
      Out.String(" ")
    END
  END;
  Out.Ln
END PrintFactors;
```

The variable `i` is local to `PrintFactors`. It does not conflict with any variable named `i` elsewhere in the program — each has its own scope.

Local variables always have undefined values at the start of a procedure call. Never assume a local variable holds zero or any other particular value — always initialise it before use, just as you would a module-level variable.

A procedure can have its own `CONST`, `TYPE`, and `VAR` sections, all following the same ordering rules as at module level.

13.6 Scope and Visibility

The **scope** of an identifier is the region of the program text where it can be used. Oberon-07's scope rules are straightforward.

A module-level declaration (in the module's `CONST`, `TYPE`, or `VAR` section) is visible throughout the entire module — in all procedures and in the main `BEGIN`...`END` block.

A procedure-level declaration (local to a procedure) is visible only within that procedure. It cannot be seen from outside, and it does not affect anything with the same name at module level or in other procedures.

If a local identifier has the same name as a module-level identifier, the local one takes precedence — it **shadows** the outer one within the procedure. This is legal but should be avoided, as it makes the code harder to read.

The practical rule is simple: procedures communicate with the rest of the program through their

parameters and, when necessary, through module-level variables. Everything else should be local.

Example 13.6

```
MODULE ScopeDemo;
  IMPORT Out;
VAR
  x : INTEGER;    (* module-level: visible everywhere *)

PROCEDURE ShowX;
VAR
  x : INTEGER;    (* local: shadows the module-level x inside this procedure *)
BEGIN
  x := 99;        (* sets the local x, not the module-level x *)
  Out.String("Inside ShowX, x = ");
  Out.Int(x, 0);
  Out.Ln
END ShowX;

BEGIN
  x := 42;
  ShowX;          (* prints 99 *)
  Out.String("After ShowX,  x = ");
  Out.Int(x, 0);  (* prints 42 -- module-level x is unchanged *)
  Out.Ln
END ScopeDemo.
```

Output:

```
Inside ShowX, x = 99
After ShowX,  x = 42
```

Another major benefit of using procedures, in addition to those given at the beginning of this chapter, is that they facilitate **information hiding**, one of the most important habits a programmer can develop. A well-designed procedure is like a sealed unit: it exposes a clear interface — its name and parameters — while everything it does internally stays hidden from the rest of the program. The caller doesn't need to know, and shouldn't need to care, what local variables a procedure uses or how it arrives at its result; it only needs to know what to pass in and what to expect back.

This is the same idea you rely on every time you use a library procedure like `Out.Int` — you have no idea what local variables it declares internally, and you don't need to. Global (module-level) variables work against this. Every module-level variable is a piece of shared state that any procedure in the module can read or silently change, which means that to understand what one procedure does, you may end up having to read every other procedure that touches the same variable. As a program grows, this turns into a tangle of hidden dependencies where a change in one place causes an unexpected bug somewhere else entirely.

Local variables and parameters don't have this problem: a local variable cannot leak out and cannot be affected by code elsewhere, so a procedure that only uses locals and parameters can be understood — and debugged — entirely on its own. For this reason, treat module-level variables as a last resort, reserved for state that genuinely needs to persist across procedure calls or be shared by several procedures. Everything else — every loop counter, every intermediate result, every temporary value — belongs inside the procedure that uses it.

13.7 Nested Procedures

A procedure may be declared *inside* another procedure. This is called a **nested procedure**, and it is only visible within the procedure that contains it — it cannot be called from outside.

Nested procedures follow stricter scope rules than you might expect. A nested procedure can see the module's global variables, and it can see its own parameters and local variables. However, it **cannot** see the local variables or parameters of the outer procedure that contains it. If it needs data from the outer procedure, that data must be passed as an explicit parameter.

Here is a demonstration:

Example 13.7

```
MODULE NestedDemo;
  IMPORT Out;
VAR
  globalCount : INTEGER;

PROCEDURE OuterProc(x : INTEGER);
VAR
  outerLocal : INTEGER;

  PROCEDURE InnerProc(y : INTEGER);
  VAR
    innerLocal : INTEGER;
  BEGIN
    innerLocal  := y * 2;
    INC(globalCount);       (* ALLOWED: can access global variables  *)
    (* outerLocal := 10;    -- NOT ALLOWED: cannot see outer locals  *)
    (* x := 5;              -- NOT ALLOWED: cannot see outer params  *)
    Out.String("Inner result: ");
    Out.Int(innerLocal, 0);
    Out.Ln
  END InnerProc;

BEGIN
  outerLocal := x;
  Out.String("OuterProc called with x = ");
  Out.Int(x, 0);
  Out.Ln;
```

```
  InnerProc(outerLocal);   (* passes outerLocal as argument to inner *)
  Out.String("Global count: ");
  Out.Int(globalCount, 0);
  Out.Ln
END OuterProc;

BEGIN
  globalCount := 0;
  OuterProc(5)
END NestedDemo.
```

Output:

```
OuterProc called with x = 5
Inner result: 10
Global count: 1
```

The key points are: `InnerProc` is declared within the declaration section of `OuterProc` and can only be called from within `OuterProc`. It can access the global variable `globalCount` directly. It cannot access `outerLocal` or the parameter `x` from `OuterProc` — those must be passed explicitly, which is why `outerLocal` is passed as the argument `y`.

When to Use Nested Procedures

Nested procedures are useful for helper actions that are only meaningful in the context of one specific outer procedure. In this context, a **helper** is simply an informal term for a small procedure whose sole purpose is to assist another procedure with part of its work — it has no useful meaning outside that specific context and would never be called directly from the main program. Nesting keeps the helper close to where it is used and prevents it from polluting the module's **namespace** — the collection of all names (procedures, variables, constants, types) that are visible at module level. In a large program, the module namespace can become crowded, and a procedure name that is only useful in one specific context adds clutter and risks clashing with other names. If you find that you want to call the helper from more than one procedure, that is a signal it should be promoted to a module-level procedure instead.

13.8 Open Array Parameters

In example 13.2 `label` was declared in the `PrintInt` procedure header as:

```
PROCEDURE PrintInt(label : ARRAY OF CHAR; value : INTEGER);
```

Why was the size of the array omitted from this declaration?

In Oberon-07, you cannot pass an array of a specific named size as a procedure parameter in the usual way — the compiler would require the parameter type to exactly match the declared size of every array you ever want to pass, which would be an intolerable restriction. The solution is **open**

array parameters: array parameters declared without a fixed size, using the syntax `ARRAY OF type`:

```
PROCEDURE PrintStrings(s : ARRAY OF CHAR);
BEGIN
  Out.String(s);
  Out.Ln
END PrintStrings;
```

An open array parameter accepts *any* array of the given element type, regardless of its declared size. If you need to know the size of the array when processing it inside the procedure, you *could* pass in its size using a constant or variable, but this would require another parameter and possibly another variable. A simpler and more idiomatic way is use the `LEN(s)` function; it returns the length of the array that was passed. The above procedure used a value parameter, but open array parameters can also be of the `VAR` type. The following example demonstrates all these features.

Example 13.8

```
MODULE OpenArrayDemo;
IMPORT Out;
CONST
  SmallSize = 4;
  LargeSize = 8;
VAR
  small : ARRAY SmallSize OF INTEGER;
  large : ARRAY LargeSize OF INTEGER;
  i     : INTEGER;

PROCEDURE PrintArray(a : ARRAY OF INTEGER);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO LEN(a) - 1 DO
    Out.Int(a[i], 5)
  END;
  Out.Ln
END PrintArray;

PROCEDURE ScaleArray(VAR a : ARRAY OF INTEGER; factor : INTEGER);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO LEN(a) - 1 DO
    a[i] := a[i] * factor
  END
END ScaleArray;
```

```
BEGIN
  FOR i := 0 TO SmallSize - 1 DO small[i] := (i + 1) * 10 END;
  FOR i := 0 TO LargeSize - 1 DO large[i] := (i + 1) * 5  END;
  PrintArray(small);
  PrintArray(large);
  ScaleArray(small, 2);
  ScaleArray(large, 3);
  PrintArray(small);
  PrintArray(large)
END OpenArrayDemo.
```

Output:

```
10   20   30   40
 5   10   15   20   25   30   35   40
20   40   60   80
15   30   45   60   75   90  105  120
```

13.9 Stubs and Drivers

When writing a program with several procedures, it is rarely practical to write all the procedures at once and only test after everything is complete. Errors are much easier to find and fix when each piece is tested in isolation. Two techniques support this: **stubs** and **drivers**.

Stubs

A **stub** is a minimal, placeholder version of a procedure. It has the correct name and parameter list, but its body does little or nothing useful — just enough to let the rest of the program compile and run while the real implementation is not yet written. A stub typically prints a message confirming it was called, so you can see that the call is being made correctly.

For example, suppose we are writing the `Contacts` program and want to test the main structure before implementing `SortByName`:

```
PROCEDURE SortByName(VAR contacts : ARRAY OF Contact; count : INTEGER);
BEGIN
  (* STUB: not yet implemented *)
  Out.String("[SortByName called with ");
  Out.Int(count, 0);
  Out.String(" contacts]");
  Out.Ln
END SortByName;
```

The program compiles and runs. The main block calls `SortByName` and you can confirm the call happens at the right point and with the right count — even though no actual sorting is done yet.

Once you are happy with the overall structure, you replace the stub with the real implementation.

Stubs are invaluable when building top-down: write the high-level structure first, stub out the details, verify the structure, then fill in the details one procedure at a time.

Drivers

A **driver** is the opposite: a temporary main block written specifically to test one procedure in isolation. Instead of building the full program around a procedure, you write a simple `BEGIN...END` block that calls the procedure with known inputs and prints the results, so you can verify it works correctly before integrating it into the larger program.

For example, to test the `Swap` procedure from Section 13.4 before using it in a larger sorting program:

Example 13.9

```
MODULE TestSwap;
  IMPORT Out;
VAR
  x, y : INTEGER;

PROCEDURE Swap(VAR a, b : INTEGER);
VAR
  temp : INTEGER;
BEGIN
  temp := a;
  a    := b;
  b    := temp
END Swap;

BEGIN
  (* Driver: test Swap with several known cases *)
  x := 10; y := 20; Swap(x, y);
  Out.String("10,20 -> ");
  Out.Int(x, 0); Out.String(","); Out.Int(y, 0); Out.Ln;
  x := 0; y := 0; Swap(x, y);
  Out.String("0,0   -> ");
  Out.Int(x, 0); Out.String(","); Out.Int(y, 0); Out.Ln;
  x := -5; y := 3; Swap(x, y);
  Out.String("-5,3  -> ");
  Out.Int(x, 0); Out.String(","); Out.Int(y, 0); Out.Ln
END TestSwap.
```

Output:

```
10,20 -> 20,10
0,0   -> 0,0
```

```
-5,3  -> 3,-5
```

A good driver tests typical cases, boundary cases (such as equal values or zero), and any unusual cases you can think of. Once the procedure passes all the driver tests, you can be confident in it and use it in the full program.

Using stubs and drivers together is a powerful technique: build the structure from the top down (using stubs), and build confidence in individual procedures from the bottom up (using drivers). The two approaches complement each other and together make large programs much easier to develop and debug.

13.10 Procedures as Building Blocks

The real power of procedures is not any individual feature — it is the way they allow you to build programs from clearly named, independently understandable pieces. A well-designed procedure does one thing and does it well. Its name describes what it does. Its parameters describe what it needs. Together, a set of good procedures makes a program read almost like a description of the problem it solves.

This is a qualitative shift from what we have been doing. In earlier chapters, programs were sequences of statements. With procedures, programs become compositions of named actions. The main block calls `ReadData`, `ProcessData`, and `PrintResults` — and each of those calls smaller procedures in turn. You can understand the structure at a glance, then look at each piece individually.

This is sometimes described as **procedural decomposition**: breaking a problem down into procedures, and those procedures into smaller procedures, until each piece is simple enough to write directly. It is the same stepwise refinement idea from Chapter 7, but now each refined step becomes a named procedure rather than just a comment.

13.11 Procedures and Records

Records are among the most common data structures passed to procedures, so it is worth looking at the specific considerations that arise when combining the two.

Passing a Record as a Value Parameter

A record can be passed to a procedure as a value parameter in the usual way. The procedure receives a complete copy of all the record's fields and can read them freely:

```
TYPE
  Date = RECORD
    day, month, year : INTEGER
  END;

PROCEDURE PrintDate(d : Date);
BEGIN
  Out.Int(d.day, 0);   Out.String("/");
```

```
  Out.Int(d.month, 0); Out.String("/");
  Out.Int(d.year, 0);  Out.Ln
END PrintDate;
```

Here `d` is a local copy of whatever `Date` record is passed in. Any assignment to `d.day` or `d.month` inside the procedure would affect only the copy and leave the original unchanged.

Passing a Record as a VAR Parameter

Use a `VAR` parameter when the procedure needs to modify the record's fields and have those changes visible to the caller — for example, a procedure that reads a record from the user:

```
PROCEDURE ReadDate(VAR d : Date);
BEGIN
  Out.String("Day:   "); In.Int(d.day);
  Out.String("Month: "); In.Int(d.month);
  Out.String("Year:  "); In.Int(d.year)
END ReadDate;
```

After a call `ReadDate(today)`, the fields `today.day`, `today.month`, and `today.year` hold whatever the user entered. Without `VAR`, the procedure would write into a local copy and the original `today` would be unchanged.

Using VAR for Efficiency

There is a second reason to use `VAR` with records that has nothing to do with modification: **efficiency**. Passing a record by value copies all its fields. For a record with many fields — or an array of records — this copying can be expensive. Passing by `VAR` avoids the copy entirely, because the procedure receives a reference to the original rather than a duplicate.

The convention in Oberon-07 is therefore: if a procedure takes a record parameter and does not need to modify it, but the record is large, use `VAR` anyway to avoid unnecessary copying. It is worth adding a comment to make the intent clear:

```
PROCEDURE PrintContact(VAR c : Contact);    (* VAR for efficiency, not modification *)
BEGIN
  Out.String("Name:  "); Out.String(c.name);  Out.Ln;
  Out.String("Phone: "); Out.String(c.phone); Out.Ln;
  Out.String("Age:   "); Out.Int(c.age, 0);   Out.Ln
END PrintContact;
```

Passing Individual Fields

You can also pass individual fields of a record as arguments, just as you would pass any variable of the field's type:

```
VAR
  today : Date;
...
Out.Int(today.year, 0);          (* passing a field value directly *)
PrintInt("Year: ", today.year);  (* passing to a procedure         *)
```

This is useful when a procedure only needs one or two fields rather than the whole record.

Example 13.11: Date Procedures

Here is a complete program that uses both a `ReadDate` and a `PrintDate` procedure, demonstrating the two parameter styles for records:

```
MODULE DateDemo;
  IMPORT In, Out;
TYPE
  Date = RECORD
    day, month, year : INTEGER
  END;
VAR
  d1, d2 : Date;

PROCEDURE ReadDate(VAR d : Date);
BEGIN
  Out.String("Day:   "); In.Int(d.day);
  Out.String("Month: "); In.Int(d.month);
  Out.String("Year:  "); In.Int(d.year)
END ReadDate;

PROCEDURE PrintDate(VAR d : Date);    (* VAR for efficiency *)
BEGIN
  IF d.day   < 10 THEN Out.String("0") END; Out.Int(d.day,   0);
  Out.String("/");
  IF d.month < 10 THEN Out.String("0") END; Out.Int(d.month, 0);
  Out.String("/");
  Out.Int(d.year, 0);
  Out.Ln
END PrintDate;

PROCEDURE IsEarlier(VAR a, b : Date) : BOOLEAN;
VAR
  result : BOOLEAN;
BEGIN
  IF a.year # b.year THEN
    result := a.year < b.year
  ELSIF a.month # b.month THEN
```

```
    result := a.month < b.month
  ELSE
    result := a.day < b.day
  END;
  RETURN result
END IsEarlier;

BEGIN
  Out.String("Enter first date:");  Out.Ln; ReadDate(d1);
  Out.String("Enter second date:"); Out.Ln; ReadDate(d2);
  Out.Ln;
  Out.String("Date 1: "); PrintDate(d1);
  Out.String("Date 2: "); PrintDate(d2);
  Out.Ln;
  IF IsEarlier(d1, d2) THEN
    Out.String("Date 1 is earlier.")
  ELSIF IsEarlier(d2, d1) THEN
    Out.String("Date 2 is earlier.")
  ELSE
    Out.String("The dates are equal.")
  END;
  Out.Ln
END DateDemo.
```

Sample run:

```
Enter first date:
Day:   15
Month: 6
Year:  2024
Enter second date:
Day:   3
Month: 11
Year:  2023

Date 1: 15/06/2024
Date 2: 03/11/2023

Date 2 is earlier.
```

This example demonstrates all three patterns: `ReadDate` uses `VAR` because it modifies the record; `PrintDate` uses `VAR` for efficiency even though it does not modify the record; and `IsEarlier` uses `VAR` on both parameters for efficiency, returning a `BOOLEAN` result through the function return value rather than a `VAR` parameter. Note that `IsEarlier` is a function procedure — covered fully in Chapter 14 — but is included here to show the natural combination of records and function procedures.

Records in Arrays Passed to Procedures

When an array of records is passed to a procedure, it should be declared as a `VAR` parameter — both to avoid copying the entire array and to allow the procedure to modify individual records if needed:

```
PROCEDURE PrintAll(VAR contacts : ARRAY OF Contact; count : INTEGER);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO count - 1 DO
    PrintContact(contacts[i])    (* passes each element by VAR *)
  END
END PrintAll;
```

Each element `contacts[i]` is passed to `PrintContact` as a `VAR` parameter. This is valid because an indexed element of an array is a variable — it has a specific memory location — and `VAR` parameters require a variable, not a value.

13.12 Procedures and Pointers

Pointers can be passed to procedures like any other value, but because a pointer's value is an address rather than data, there are two different things a procedure might be asked to change: the record the pointer refers to, or the pointer itself — which record it refers to, or whether it refers to one at all. Which of these you need determines whether an ordinary value parameter is enough, or whether `VAR` is required.

We will use the `Node` type from Chapter 11 throughout:

```
TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD
    value : INTEGER;
    next  : Node
  END;
```

Passing a Pointer as a Value Parameter

When a pointer is passed as a value parameter, the procedure receives a copy of the pointer. That copy is a separate variable from the caller's — but it still refers to the very same record, since copying a pointer copies the address, not the data it points to. Using the luggage-tag analogy from Chapter 11: the procedure gets a duplicate tag, but the tag still routes to the caller's suitcase.

This has an important consequence. Reading or modifying the fields of the record through the local copy affects the caller's record too, because there is only one record and both tags point to it. If this procedure:

```
PROCEDURE Increment(n : NodePtr);
BEGIN
  n.value := n.value + 1
END Increment;
```

is called with `Increment(head)`, the procedure increments `head.value`, even though `n` is an ordinary value parameter, not a `VAR` parameter. Passing a `NodePtr` by value does not copy the `Node` it points to, only the address. Contrast this with Section 13.11, where passing a `Date` record by value copies every field, and changes inside the procedure never reach the caller.

What a value parameter *cannot* do is change which record the caller's pointer variable refers to. If the procedure assigns something to `n` itself, or calls `NEW(n)` to allocate a fresh record, only the local copy is affected:

```
PROCEDURE BadMakeNode(n : NodePtr; value : INTEGER);
BEGIN
  NEW(n);  (* n now points to a new record, but only the LOCAL copy of n *)
  n.value := value;
  n.next := NIL
END BadMakeNode;
```

After `BadMakeNode(head, 10)`, the caller's `head` is exactly what it was before the call — still `NIL`, if that's what it started as. `NEW(n)` replaced the local copy's address; it had no way to reach back and change the caller's variable.

Passing a Pointer as a VAR Parameter

To let a procedure allocate a new record and have the *caller's* pointer variable point to it, the pointer must be passed as a `VAR` parameter.

When this is done, the procedure creates something on the heap and hands back access to it, so that the new record outlives the procedure call that created it

Think of it like a hotel concierge issuing you a room key. The room doesn't belong to the concierge, and it doesn't vanish when you walk away from the desk — the key is just a way of reaching it later. A procedure can do the same thing with `NEW`: create a record on the heap and pass the "key" back to the caller, long after the procedure itself has finished running.

Example 13.12

```
MODULE BuildList;
  IMPORT Out;
TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD
    value : INTEGER;
    next  : NodePtr
```

```
  END;
VAR
  head : NodePtr;

PROCEDURE MakeNode(VAR n : NodePtr; value : INTEGER);
BEGIN
  NEW(n);
  n.value := value;
  n.next := NIL
END MakeNode;

PROCEDURE PrintList(n : NodePtr);  (* value parameter: we are only reading *)
BEGIN
  WHILE n # NIL DO
    Out.Int(n.value, 0);
    Out.String(" ");
    n := n.next
  END;
  Out.Ln
END PrintList;

BEGIN
  MakeNode(head, 10);
  MakeNode(head.next, 20);
  MakeNode(head.next.next, 30);
  PrintList(head)
END BuildList.
```

Output:

```
10 20 30
```

Each call to `MakeNode` passes a field of type `NodePtr` — `head`, then `head.next`, then `head.next.next` — as the `VAR` argument. A field selector denotes a specific variable, exactly as an array element does, so it is a legal `VAR` argument. Each call allocates a fresh record and writes its address into that specific field, extending the list one node at a time.

`PrintList`, by contrast, only needs to *read* the list — it never changes which record any pointer refers to from the caller's point of view, so a value parameter is enough. Inside the procedure, `n := n.next` walks the local copy of the pointer forward; whatever `n` is reassigned to cannot affect the caller's `head`, since `n` is a value parameter.

In summary, when a pointer is passed as a **value parameter**, the procedure gets its own copy of the address — a duplicate luggage tag — but that tag still routes to the same suitcase as the caller's. This means you can freely read and modify the fields of the record through the parameter (`n.value := n.value + 1` *does* change the caller's data), and you can safely reassign the parameter itself inside the procedure (to walk through a list, say) without any risk of disturbing the caller's variable, since it's a

separate copy from the moment the call is made. What a value parameter can never do is make the caller's pointer variable point somewhere new — in particular, calling `NEW` on a value parameter only allocates a record for the local copy, which is discarded when the procedure returns.

Use a value parameter whenever the procedure only needs to read or modify an existing record through the pointer. Use a `VAR` parameter whenever the procedure needs to change *what the caller's own variable points to* — most commonly, allocating a new record with `NEW` and having that new record persist after the procedure returns, as in the `MakeNode`procedure above. The rule of thumb: value parameter for working with a record that already exists; VAR parameter for handing the caller a pointer to something new.

13.13 Refactoring the Contact Book

In Chapter 10, the `Contacts` program was a single long sequence of statements in the main block. Let us refactor it — restructure it using procedures — without changing what it does. The goal is to make the structure clearer and each piece independently readable.

We identify the following natural pieces: reading a single contact from the user, printing a single contact, printing a separator line, sorting the array by name, and printing all contacts. Each becomes a procedure.

Example 13.10

```
MODULE Contacts;
  IMPORT In, Out;
CONST
  MaxContacts = 5;
  NameLen     = 25;
  PhoneLen    = 15;
TYPE
  NameStr  = ARRAY NameLen  OF CHAR;
  PhoneStr = ARRAY PhoneLen OF CHAR;
  Contact  = RECORD
    name  : NameStr;
    phone : PhoneStr;
    age   : INTEGER
  END;
VAR
  book  : ARRAY MaxContacts OF Contact;
  flush : ARRAY 10 OF CHAR;
  n     : INTEGER;
  done  : BOOLEAN;

PROCEDURE PrintSeparator;
BEGIN
  Out.String("------------");
  Out.Ln
END PrintSeparator;
```

```
PROCEDURE PrintContact(VAR c : Contact);
BEGIN
  Out.String("Name:  "); Out.String(c.name);  Out.Ln;
  Out.String("Phone: "); Out.String(c.phone); Out.Ln;
  Out.String("Age:   "); Out.Int(c.age, 0);   Out.Ln;
  PrintSeparator
END PrintContact;

PROCEDURE ReadContact(VAR c : Contact; VAR blank : BOOLEAN);
BEGIN
  Out.String("Name:  ");
  In.Line(c.name);
  IF c.name[0] = 0X THEN
    blank := TRUE
  ELSE
    Out.String("Phone: ");
    In.Name(c.phone);
    Out.String("Age:   ");
    In.Int(c.age);
    In.Line(flush);
    Out.Ln;
    blank := FALSE
  END
END ReadContact;

PROCEDURE SortByName(VAR contacts : ARRAY OF Contact; count : INTEGER);
VAR
  temp         : Contact;
  i, j, minPos : INTEGER;
BEGIN
  FOR i := 0 TO count - 2 DO
    minPos := i;
    FOR j := i + 1 TO count - 1 DO
      IF contacts[j].name < contacts[minPos].name THEN
        minPos := j
      END
    END;
    IF minPos # i THEN
      temp             := contacts[i];
      contacts[i]      := contacts[minPos];
      contacts[minPos] := temp
    END
  END
END SortByName;
```

```
PROCEDURE PrintAll(VAR contacts : ARRAY OF Contact; count : INTEGER);
VAR
  i : INTEGER;
BEGIN
  PrintSeparator;
  FOR i := 0 TO count - 1 DO
    PrintContact(contacts[i])
  END
END PrintAll;

BEGIN
  n    := 0;
  done := FALSE;
  Out.String("Enter contacts (up to ");
  Out.Int(MaxContacts, 0);
  Out.String("). Press Enter with a blank name to finish.");
  Out.Ln;
  WHILE (n < MaxContacts) & ~done DO
    ReadContact(book[n], done);
    IF ~done THEN INC(n) END
  END;
  IF n = 0 THEN
    Out.String("No contacts entered.");
    Out.Ln
  ELSE
    Out.Ln;
    Out.String("Contact List");
    Out.Ln;
    PrintAll(book, n);
    SortByName(book, n);
    Out.Ln;
    Out.String("Sorted alphabetically:");
    Out.Ln;
    PrintAll(book, n)
  END
END Contacts.
```

Compare this main block with the original in Chapter 10. The logic is now expressed in five lines — `ReadContact`, `PrintAll`, `SortByName`, `PrintAll` again — and a reader can understand the overall structure immediately without reading a single statement of detail. The details are in the procedures, where they can be read and understood one piece at a time.

A few things are worth noting about the procedure designs. `ReadContact` takes the contact as a `VAR` parameter because it writes into it, and also takes a `VAR BOOLEAN` parameter `ok` to signal whether a valid contact was entered rather than a blank name. `PrintContact` takes the contact as a `VAR` parameter not because it modifies it, but because passing a record by value would copy all its fields unnecessarily — using `VAR` is more efficient for large records even when no modification is intended.

`SortByName` takes the array as a `VAR` parameter because it modifies the array in place.

Summary

- A **procedure** is a named block of declarations and statements that performs a specific task. It is declared before the module's `BEGIN` and called by name. The procedure name is repeated after the closing `END`.
- **Parameters** allow data to be passed into a procedure. They are declared in a parameter list after the procedure name.
- A **value parameter** receives a copy of the argument. The procedure cannot modify the caller's variable through it.
- A **VAR parameter** receives a reference to the caller's variable. Changes made through a `VAR` parameter are visible in the caller. Use `VAR` when the procedure needs to modify the argument, or to avoid copying large structures.
- **Local variables** are declared inside a procedure and exist only while it runs. They are independent of any module-level variables with the same name. An identifier declared locally **shadows** any module-level identifier with the same name within that procedure.
- A **nested procedure** is declared inside another procedure and is only visible within it. It can access global variables but not the local variables or parameters of its enclosing procedure — those must be passed explicitly.
- **Open array parameters** (`ARRAY OF type`) accept arrays of any size. `LEN` applied to an open array parameter returns the actual size of the array passed by the caller.
- A **stub** is a placeholder procedure with the correct signature but a minimal body. It allows a program to compile and run while a procedure's real implementation is not yet written.
- A **driver** is a temporary main block that tests one procedure in isolation with known inputs. Used together, stubs and drivers support building programs top-down while testing bottom-up.
- **Procedural decomposition** — breaking a program into well-named procedures — is the primary technique for managing complexity. A good procedure does one thing, has a clear name, and communicates through its parameters.

Exercises

13.1 Write a procedure `PrintStars(n : INTEGER)` that prints `n` asterisk characters on a single line followed by a newline. Write a complete program that calls it several times with different values to produce a simple bar chart. Use a driver approach: test `PrintStars` with a few values before embedding it in the bar chart logic.

13.2 Write a procedure `FindMax(a, b : INTEGER; VAR result : INTEGER)` that stores the larger of `a` and `b` in `result`. Write a driver program that tests it with several pairs of values, including equal values and negative numbers.

13.3 Write a procedure `Swap(VAR a, b : REAL)` that swaps two real values. Write a driver that tests it with several cases, then use it in a program that reads two reals, swaps them, and prints the results.

13.4 Write a procedure `ReadDate(VAR day, month, year : INTEGER)` that prompts the user for a day, month, and year and reads each one. Write a corresponding procedure `PrintDate(day, month, year`

`: INTEGER)` that prints the date in the format `DD/MM/YYYY`, padding day and month with a leading zero if needed. Write a program that uses both procedures to read and print three dates.

13.5 Write a procedure `PrintArray(a : ARRAY OF INTEGER; n : INTEGER)` that prints the first `n` elements of array `a`, one per line, right-justified in a field of width 6. Write a program that fills an array with the first 10 squares and calls the procedure to print them.

13.6 Write a procedure `ReverseArray(VAR a : ARRAY OF INTEGER; n : INTEGER)` that reverses the first `n` elements of `a` in place. Write a driver to test it thoroughly — try an array of one element, an even number of elements, and an odd number of elements. Then use it in a program that fills an array with values 1 to 8, reverses it, and prints the result.

13.7 Refactor the `Stats` program from Chapter 8 (Section 8.8) using procedures. Write separate procedures for reading the data into the array, computing the statistics (minimum, maximum, sum, mean), and printing the results. The main block should do nothing more than call these three procedures in sequence. Write stubs for the compute and print procedures first and verify the overall structure before implementing them.

13.8 Extend the `Contacts` program from Section 13.11 with a procedure `FindByName` that takes the array, the count, and a name string, and prints the contact with that name if found, or a "not found" message if not. Write a stub for `FindByName` first, verify it is called correctly, then implement it.

Chapter 14 — Function Procedures

In Chapter 13 we learned how procedures can take inputs through parameters and produce effects — printing output, modifying variables, sorting arrays. But sometimes what we want from a procedure is not an effect but a **value**: compute the larger of two numbers, calculate the area of a circle, determine whether a string is a palindrome. For these cases, Oberon-07 provides **function procedures** — procedures that compute and return a single result.

You have been using function procedures throughout this book. `ORD`, `CHR`, `ABS`, `LEN`, `FLT`, `FLOOR`, `ODD` — all of these are function procedures, built into the language. `Strings.Length` and `Strings.Pos` are function procedures from the `Strings` module. Now you will learn to write your own.

14.1 The RETURN Statement

A function procedure is distinguished from an ordinary procedure in two ways: its declaration includes a result type after the parameter list, and its body ends with a `RETURN` statement that specifies the value to be returned.

The syntax is:

```
PROCEDURE name(parameters) : ResultType;
  (* local declarations *)
BEGIN
  (* statements *)
  RETURN expression
END name;
```

The `RETURN` statement must appear exactly once, at the end of the procedure body. Unlike some languages, Oberon-07 does not permit multiple `RETURN` statements scattered through the body — there is one, and it is the last thing before `END`.

Here is a simple example:

```
PROCEDURE Square(n : INTEGER) : INTEGER;
BEGIN
  RETURN n * n
END Square;
```

This function takes an integer and returns its square. It is called by using it as part of an expression, just like the predefined functions:

```
Out.Int(Square(5), 0);       (* prints 25 *)
Out.Int(Square(12), 0);      (* prints 144 *)
x := Square(a) + Square(b);  (* x gets a² + b²  *)
```

A function procedure call is an expression that yields a value — it can appear anywhere an

expression of the appropriate type can appear.

14.2 Result Types

The result type of a function procedure can be any basic type: `INTEGER`, `REAL`, `BOOLEAN`, `CHAR`, or `SET`. It can also be a pointer type (see section 14.4). However, it cannot be a record type or an array type — if you need to return a record or array, you must use an ordinary procedure with a `VAR` parameter instead.

This restriction is a consequence of Oberon-07's design. Returning a large structure by value would require copying it, and the language prefers to make such operations explicit through `VAR` parameters.

14.3 Simple Function Procedures

Here is a program with several function procedures that illustrate the range of result types:

Example 14.1

```
MODULE FuncDemo;
  IMPORT Out;
VAR
  a, b : INTEGER;

PROCEDURE Max(a, b : INTEGER) : INTEGER;
VAR
  result : INTEGER;
BEGIN
  IF a > b THEN
    result := a
  ELSE
    result := b
  END;
  RETURN result
END Max;

PROCEDURE IsEven(n : INTEGER) : BOOLEAN;
BEGIN
  RETURN n MOD 2 = 0
END IsEven;

PROCEDURE InitialOf(name : ARRAY OF CHAR) : CHAR;
BEGIN
  RETURN name[0]
END InitialOf;

BEGIN
  a := 17;
```

```
  b := 42;
  Out.String("Max(17, 42) = ");
  Out.Int(Max(a, b), 0);
  Out.Ln;
  Out.String("IsEven(17)  = ");
  Out.Int(ORD(IsEven(a)), 0);
  Out.Ln;
  Out.String("IsEven(42)  = ");
  Out.Int(ORD(IsEven(b)), 0);
  Out.Ln;
  Out.String("Initial of 'Oberon' = ");
  Out.Char(InitialOf("Oberon"));
  Out.Ln
END FuncDemo.
```

Output:

```
Max(17, 42) = 42
IsEven(17)  = 0
IsEven(42)  = 1
Initial of 'Oberon' = O
```

The `Max` procedure shows the standard pattern for conditional results: use a local variable `result`, assign to it in the `IF`/`ELSE` branches, and return it at the end. `IsEven` shows that the `RETURN` expression can be any expression of the right type — `n MOD 2 = 0` is a Boolean expression whose value is returned directly. `InitialOf` returns a `CHAR` — the first character of the string.

14.4 Returning Pointers from Function Procedures

Section 14.2 mentioned that a function procedure's result type can be a pointer type, even though it cannot be a record or array type. This is genuinely useful: it gives us a second way to write a procedure that creates a new record and hands it back to the caller — an alternative to the `VAR`-parameter version from chapter 13.

Here is the `MakeNode` procedure from chapter 13 example 13.12 rewritten as a function procedure:

```
PROCEDURE MakeNode(value : INTEGER) : NodePtr;
VAR
  n : NodePtr;
BEGIN
  NEW(n);
  n.value := value;
  n.next := NIL;
  RETURN n
END MakeNode;
```

This works because a pointer is a single value — just an address — so returning one costs nothing more than returning an `INTEGER` or a `BOOLEAN`. It is called like any function procedure, as part of an expression:

```
head := MakeNode(10);
head.next := MakeNode(20);
head.next.next := MakeNode(30);
```

Example 14.7

```
MODULE BuildListFunc;
  IMPORT Out;
TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD
    value : INTEGER;
    next  : NodePtr
  END;
VAR
  head : NodePtr;

PROCEDURE MakeNode(value : INTEGER) : NodePtr;
VAR
  n : NodePtr;
BEGIN
  NEW(n);
  n.value := value;
  n.next := NIL;
  RETURN n
END MakeNode;

PROCEDURE PrintList(n : NodePtr);
BEGIN
  WHILE n # NIL DO
    Out.Int(n.value, 0);
    Out.String(" ");
    n := n.next
  END;
  Out.Ln
END PrintList;

BEGIN
  head := MakeNode(10);
  head.next := MakeNode(20);
  head.next.next := MakeNode(30);
```

```
  PrintList(head)
END BuildListFunc.
```

Output:

```
10 20 30
```

This is the same list-building task as Example 13.12, solved a different way. There, `MakeNode` filled in the caller's pointer variable through a `VAR` parameter; here, it builds the record locally and returns the pointer directly, and the caller assigns the result wherever it's needed — to `head`, or to a field like `head.next`.

Which is clearer is partly a matter of taste, but there's a practical difference worth noticing: A `VAR` parameter needs something the compiler can identify as an actual storage location — a variable, a field, or an array element — because the procedure is going to write into it. You can't pass it an expression, a literal, or the result of another function call, because there's nowhere for the result to be written back to. For example, `MakeNode(head, 10)` is fine because `head` is a variable, but something like `MakeNode(SomeExpression(), 10)` wouldn't compile because `SomeExpression()`'s result isn't a variable you could write into. The function-returning version has no such restriction, because a function result is just an ordinary value, and ordinary values can appear anywhere an expression can. So you could write something like:

```
PrintList(MakeNode(10))
```

passing the new node straight into another procedure without ever storing it in a variable first — no intermediate storage is needed. With the `VAR`-parameter version, you're always required to have a variable or field on hand — even if it's one you don't otherwise need — purely so `MakeNode` has something to write the pointer into.

14.5 Predefined Function Procedures Revisited

Now that you understand function procedures, the predefined functions from earlier chapters should feel more familiar. They are simply function procedures built into the language. For reference, here are the ones we have used:

Function	Parameter type	Result type	Description
`ABS(x)`	numeric	same type	Absolute value
`ODD(x)`	`INTEGER`	`BOOLEAN`	`TRUE` if `x` is odd
`LEN(a)`	array	`INTEGER`	Number of elements in array
`FLT(n)`	`INTEGER`	`REAL`	Convert integer to real
`FLOOR(x)`	`REAL`	`INTEGER`	Round down to nearest integer
`ORD(x)`	`CHAR`/`BOOLEAN`/`SET`	`INTEGER`	Ordinal value
`CHR(n)`	`INTEGER`	`CHAR`	Character with given ordinal

These follow the same rules as user-defined function procedures: they appear in expressions, they take arguments of specified types, and they return a single value. The only difference is that they are provided by the language itself rather than declared by the programmer.

14.6 Nested Function Procedures

Just as ordinary procedures can be nested inside other procedures (Chapter 13), function procedures can be nested too. A nested function procedure follows exactly the same scope rules: it can see global variables and its own parameters and local variables, but it cannot see the local variables or parameters of its enclosing procedure. Data from the outer procedure must be passed explicitly as arguments.

Example 14.2

```
MODULE NestedFuncDemo;
  IMPORT Out;
VAR
  globalOffset : INTEGER;

PROCEDURE OuterMath(x, y : INTEGER) : INTEGER;

  PROCEDURE InnerCalculate(a, b : INTEGER) : INTEGER;
  VAR
    innerResult : INTEGER;
  BEGIN
    innerResult := a * b;
    RETURN innerResult + globalOffset
    (* Note: x and y from OuterMath are not visible here *)
  END InnerCalculate;

BEGIN
  RETURN InnerCalculate(x, y)
END OuterMath;

BEGIN
  globalOffset := 10;
  Out.String("OuterMath(5, 4) = ");
  Out.Int(OuterMath(5, 4), 0);
  Out.Ln
END NestedFuncDemo.
```

Output:

```
OuterMath(5, 4) = 30
```

The flow is: `OuterMath` receives 5 and 4, passes them to `InnerCalculate` as `a` and `b`. `InnerCalculate` computes `5 * 4 = 20`, adds `globalOffset` (10) to get 30, and returns it. `OuterMath` receives that result and returns it directly to the caller.

`InnerCalculate` is declared inside `OuterMath` and can only be called from within it. It accesses `globalOffset` directly because that is a module-level variable. It cannot access `x` or `y` from `OuterMath` — those are passed explicitly as `a` and `b`.

14.7 Procedure Types

So far, every variable we have declared has held a data value — an integer, a string, a record. But in Oberon-07, a variable can also hold a **procedure** — specifically, it can hold a reference to a procedure that matches a particular signature. This is called a **procedure type**.

This is a powerful and somewhat surprising idea, so let us build up to it carefully.

The Concept

Suppose you want to write a procedure that applies some operation to each element of an array. The "apply to each element" part is always the same — loop through the array, visit each element. But the operation itself varies: sometimes you want to double each element, sometimes square it, sometimes set it to zero.

Without procedure types, you would need a separate procedure for each combination: `DoubleAll`, `SquareAll`, `ZeroAll`. With procedure types, you can write one `ApplyToAll` procedure that takes the operation as a parameter — and you can pass different operations to it at different times.

The key insight is: **a procedure can be passed as an argument, just like a value**. A procedure type describes what kind of procedure can be passed — its parameter types and result type.

Declaring a Procedure Type

A procedure type is declared in the `TYPE` section:

```
TYPE
  IntOp = PROCEDURE(x : INTEGER) : INTEGER;
```

This declares `IntOp` as the type of any function procedure that takes one `INTEGER` parameter and returns an `INTEGER`. A variable of this type can hold any such procedure:

```
VAR
  op : IntOp;
```

Assigning and Calling

You assign a procedure to a procedure variable by writing the procedure's name (without parentheses or arguments — just the name):

```
PROCEDURE Double(x : INTEGER) : INTEGER;
BEGIN
  RETURN x * 2
END Double;

PROCEDURE Square(x : INTEGER) : INTEGER;
BEGIN
  RETURN x * x
END Square;

VAR
  op : IntOp;
...
op := Double;        (* op now holds a reference to Double *)
Out.Int(op(5), 0);   (* calls Double(5), prints 10        *)
op := Square;        (* op now holds a reference to Square *)
Out.Int(op(5), 0);   (* calls Square(5), prints 25        *)
```

Calling `op(5)` calls whatever procedure `op` currently refers to. This is **indirect procedure call** — the procedure being called is not fixed at compile time but determined at runtime by the value of `op`.

An Important Restriction

There is one constraint to be aware of: when passing a procedure as an argument or assigning it to a procedure variable, the procedure must be declared at **module level** (globally). Locally declared procedures and nested procedures cannot be assigned to procedure variables. This is a deliberate restriction in Oberon-07's type system.

Example 14.3: Applying an Operation to an Array

Here is the `ApplyToAll` idea made concrete:

```
MODULE ApplyDemo;
  IMPORT Out;
CONST
  N = 5;
TYPE
  IntOp = PROCEDURE(x : INTEGER) : INTEGER;
VAR
  data : ARRAY N OF INTEGER;
  i    : INTEGER;

PROCEDURE Double(x : INTEGER) : INTEGER;
BEGIN
  RETURN x * 2
END Double;
```

```
PROCEDURE Square(x : INTEGER) : INTEGER;
BEGIN
  RETURN x * x
END Square;

PROCEDURE Negate(x : INTEGER) : INTEGER;
BEGIN
  RETURN -x
END Negate;

PROCEDURE ApplyToAll(VAR a : ARRAY OF INTEGER; n : INTEGER; op : IntOp);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO n - 1 DO
    a[i] := op(a[i])
  END
END ApplyToAll;

PROCEDURE PrintArray(a : ARRAY OF INTEGER; n : INTEGER);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO n - 1 DO
    Out.Int(a[i], 6)
  END;
  Out.Ln
END PrintArray;

BEGIN
  FOR i := 0 TO N - 1 DO
    data[i] := i + 1
  END;
  Out.String("Original: "); PrintArray(data, N);
  ApplyToAll(data, N, Double);
  Out.String("Doubled:  "); PrintArray(data, N);
  FOR i := 0 TO N - 1 DO
    data[i] := i + 1
  END;
  ApplyToAll(data, N, Square);
  Out.String("Squared:  "); PrintArray(data, N);
  FOR i := 0 TO N - 1 DO
    data[i] := i + 1
  END;
  ApplyToAll(data, N, Negate);
```

```
  Out.String("Negated:  "); PrintArray(data, N)
END ApplyDemo.
```

Output:

```
Original:      1     2     3     4     5
Doubled:       2     4     6     8    10
Squared:       1     4     9    16    25
Negated:      -1    -2    -3    -4    -5
```

`ApplyToAll` does not know or care which operation it is applying. It simply calls `op(a[i])` for each element. By passing `Double`, `Square`, or `Negate` as the third argument, the caller decides the operation. The same `ApplyToAll` procedure serves all three cases — and any future operation you care to add, as long as it matches the `IntOp` signature.

Example 14.4: A Comparison Procedure Type

Procedure types are not limited to functions that transform values. A very common use is for **comparison** procedures — functions that take two values and return a `BOOLEAN` indicating their relative order. This allows algorithms like sorting to work with any ordering criterion.

Consider a sort that can go either ascending or descending depending on a comparison procedure passed to it:

```
MODULE FlexSort;
  IMPORT Out;
CONST
  N = 6;
TYPE
  CompareProc = PROCEDURE(a, b : INTEGER) : BOOLEAN;
VAR
  data : ARRAY N OF INTEGER;

PROCEDURE LessThan(a, b : INTEGER) : BOOLEAN;
BEGIN
  RETURN a < b
END LessThan;

PROCEDURE GreaterThan(a, b : INTEGER) : BOOLEAN;
BEGIN
  RETURN a > b
END GreaterThan;

PROCEDURE Sort(VAR a : ARRAY OF INTEGER; n : INTEGER; before : CompareProc);
VAR
  i, j, minPos, temp : INTEGER;
```

```
BEGIN
  FOR i := 0 TO n - 2 DO
    minPos := i;
    FOR j := i + 1 TO n - 1 DO
      IF before(a[j], a[minPos]) THEN
        minPos := j
      END
    END;
    IF minPos # i THEN
      temp      := a[i];
      a[i]      := a[minPos];
      a[minPos] := temp
    END
  END
END Sort;

PROCEDURE PrintArray(a : ARRAY OF INTEGER; n : INTEGER);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO n - 1 DO
    Out.Int(a[i], 5)
  END;
  Out.Ln
END PrintArray;

BEGIN
  data[0] := 34;  data[1] := 7;  data[2] := 89;
  data[3] := 12;  data[4] := 56; data[5] := 3;
  Out.String("Original:   "); PrintArray(data, N);
  Sort(data, N, LessThan);
  Out.String("Ascending:  "); PrintArray(data, N);
  Sort(data, N, GreaterThan);
  Out.String("Descending: "); PrintArray(data, N)
END FlexSort.
```

Output:

```
Original:      34    7   89   12   56    3
Ascending:      3    7   12   34   56   89
Descending:    89   56   34   12    7    3
```

The `Sort` procedure does not contain the words “ascending” or “descending” anywhere. It simply calls `before(a[j], a[minPos])` to ask: “should `a[j]` come before `a[minPos]`?” When `before` is `LessThan`, smaller values come first; when it is `GreaterThan`, larger values come first. The sort algorithm is written once and works for both orderings — and any other ordering you define, such as

sorting by absolute value or by distance from some target number.

Example 14.5: A Procedure Variable as a Dispatcher

A procedure variable can also act as a **dispatcher** — a way of selecting which procedure to call based on a runtime decision, without a long `IF`/`ELSIF` chain. Here is a simple calculator that uses a procedure type to dispatch to the right arithmetic operation:

```
MODULE Dispatch;
  IMPORT In, Out;
TYPE
  Operation = PROCEDURE(a, b : INTEGER) : INTEGER;
VAR
  a, b   : INTEGER;
  op     : Operation;
  choice : INTEGER;

PROCEDURE Add(a, b : INTEGER) : INTEGER;
BEGIN RETURN a + b END Add;

PROCEDURE Subtract(a, b : INTEGER) : INTEGER;
BEGIN RETURN a - b END Subtract;

PROCEDURE Multiply(a, b : INTEGER) : INTEGER;
BEGIN RETURN a * b END Multiply;

BEGIN
  Out.String("Enter two integers: "); Out.Ln;
  In.Int(a); In.Int(b);
  Out.String("1=Add  2=Subtract  3=Multiply: "); Out.Ln;
  In.Int(choice);
  IF choice = 1 THEN
    op := Add
  ELSIF choice = 2 THEN
    op := Subtract
  ELSE
    op := Multiply
  END;
  Out.String("Result: ");
  Out.Int(op(a, b), 0);
  Out.Ln
END Dispatch.
```

The `IF`/`ELSIF` block assigns the chosen procedure to `op`. After that, a single call `op(a, b)` performs whatever operation was selected. In a larger program this pattern scales well: new operations can be added by writing a new procedure and adding one branch to the assignment block, without changing the call site at all. We will revisit this pattern in Chapter 21, where it becomes the foundation for

Oberon-07's approach to object-oriented programming.

14.8 A Library of Maths Utilities

Let us bring together function procedures in a small library of mathematical utilities. The program computes several statistics about an array of real numbers, using function procedures for each calculation.

Example 14.6

```
MODULE MathUtils;
  IMPORT In, Out;
CONST
  MaxN = 20;
VAR
  data  : ARRAY MaxN OF REAL;
  n     : INTEGER;
  x     : REAL;

PROCEDURE Min(a, b : REAL) : REAL;
VAR
  result : REAL;
BEGIN
  IF a < b THEN result := a ELSE result := b END;
  RETURN result
END Min;

PROCEDURE Max(a, b : REAL) : REAL;
VAR
  result : REAL;
BEGIN
  IF a > b THEN result := a ELSE result := b END;
  RETURN result
END Max;

PROCEDURE Sum(a : ARRAY OF REAL; n : INTEGER) : REAL;
VAR
  i   : INTEGER;
  tot : REAL;
BEGIN
  tot := 0.0;
  FOR i := 0 TO n - 1 DO
    tot := tot + a[i]
  END;
  RETURN tot
END Sum;
```

```
PROCEDURE Mean(a : ARRAY OF REAL; n : INTEGER) : REAL;
BEGIN
  RETURN Sum(a, n) / FLT(n)
END Mean;

PROCEDURE ArrayMin(a : ARRAY OF REAL; n : INTEGER) : REAL;
VAR
  i   : INTEGER;
  low : REAL;
BEGIN
  low := a[0];
  FOR i := 1 TO n - 1 DO
    low := Min(low, a[i])
  END;
  RETURN low
END ArrayMin;

PROCEDURE ArrayMax(a : ARRAY OF REAL; n : INTEGER) : REAL;
VAR
  i    : INTEGER;
  high : REAL;
BEGIN
  high := a[0];
  FOR i := 1 TO n - 1 DO
    high := Max(high, a[i])
  END;
  RETURN high
END ArrayMax;

PROCEDURE ReadData;
BEGIN
  n := 0;
  Out.String("Enter real numbers (non-number to finish):");
  Out.Ln;
  In.Real(x);
  WHILE In.Done & (n < MaxN) DO
    data[n] := x;
    INC(n);
    In.Real(x)
  END
END ReadData;

BEGIN
  ReadData;
  IF n = 0 THEN
    Out.String("No data entered.");
```

```
    Out.Ln
  ELSE
    Out.String("Count:   "); Out.Int(n, 0);                  Out.Ln;
    Out.String("Sum:     "); Out.Real(Sum(data, n), 0);       Out.Ln;
    Out.String("Mean:    "); Out.Real(Mean(data, n), 0);      Out.Ln;
    Out.String("Minimum: "); Out.Real(ArrayMin(data, n), 0);  Out.Ln;
    Out.String("Maximum: "); Out.Real(ArrayMax(data, n), 0);  Out.Ln
  END
END MathUtils.
```

With inputs 3.0, 7.5, 2.0, 9.5, 4.0:

```
Count:   5
Sum:     2.60000E+01
Mean:    5.20000E+00
Minimum: 2.00000E+00
Maximum: 9.50000E+00
```

The program illustrates a useful pattern: function procedures can call other function procedures, building more complex calculations from simpler ones. Mean calls Sum rather than duplicating its loop; ArrayMin and ArrayMax each call Min and Max to compare pairs of values. This kind of composition — each function doing one clear job, with larger functions built from smaller ones — is exactly the same principle as procedural decomposition from Chapter 13, applied to functions that return values.

Summary

- A **function procedure** returns a value. Its declaration includes a result type after the parameter list, and its body ends with a single `RETURN` statement that specifies the value to be returned.
- The `RETURN` statement appears exactly once, at the end of the procedure body. Multiple `RETURN` statements are not permitted. Use a local variable to accumulate conditional results and return it once at the end.
- The result type can be any basic type (`INTEGER`, `REAL`, `BOOLEAN`, `CHAR`, `SET`) or a pointer type. It cannot be a record or array type.
- **Nested function procedures** follow the same scope rules as nested ordinary procedures: they can access global variables but not the local variables or parameters of their enclosing procedure.
- A **procedure type** describes the signature of a procedure — its parameter types and result type. A variable of a procedure type can hold a reference to any procedure matching that signature.
- Procedure variables are assigned by writing the procedure's name (without arguments). They are called by writing the variable's name with arguments.
- Only module-level procedures can be assigned to procedure variables. Nested or local procedures cannot.
- Procedure types enable writing **generic algorithms** — procedures like `ApplyToAll` or `Sort`

that work with any operation or comparison matching the declared type, without knowing in advance which specific procedure will be used.

Exercises

14.1 Write a function procedure `Cube(n : INTEGER) : INTEGER` that returns the cube of its argument. Write a complete program that prints a table of cubes from 1 to 10 using this function.

14.2 Write a function procedure `IsPrime(n : INTEGER) : BOOLEAN` that returns `TRUE` if `n` is a prime number and `FALSE` otherwise. Write a program that uses it to print all primes from 2 to 50.

14.3 Write a function procedure `Clamp(x, lo, hi : INTEGER) : INTEGER` that returns `lo` if `x < lo`, `hi` if `x > hi`, and `x` otherwise (clamping `x` to the range `[lo, hi]`). Test it with a driver that tries values inside, below, and above the range.

14.4 Write a function procedure `CountChar(s : ARRAY OF CHAR; ch : CHAR) : INTEGER` that counts and returns the number of occurrences of character `ch` in string `s`. Test it with several strings and characters.

14.5 Declare a procedure type `Predicate = PROCEDURE(n : INTEGER) : BOOLEAN`. Write two procedures matching this type: `IsEven` and `IsPositive`. Write a procedure `CountIf(a : ARRAY OF INTEGER; n : INTEGER; test : Predicate) : INTEGER` that counts how many elements of `a` satisfy `test`. Test it by counting the even numbers and the positive numbers in an array.

14.6 Extend the `FlexSort` program from Section 14.6 to sort an array of strings (use `TYPE Word = ARRAY 20 OF CHAR` and `TYPE CompareWords = PROCEDURE(a, b : ARRAY OF CHAR) : BOOLEAN`). Write `AlphaBefore` and `AlphaAfter` comparison procedures and use them to sort a fixed array of words both alphabetically and reverse-alphabetically.

14.7 Write a function procedure `ApplyTwice(f : IntOp; x : INTEGER) : INTEGER` that applies the function `f` to `x` twice — that is, returns `f(f(x))`. Using the `Double` and `Square` procedures from Section 14.6, test what `ApplyTwice` produces for several inputs. Can you predict the results before running the program?

14.8 Extend the `MathUtils` program from Section 14.7 with a function procedure `Variance(a : ARRAY OF REAL; n : INTEGER) : REAL` that computes the variance of the data. The variance is the mean of the squared deviations from the mean: for each value `x`, compute `(x - mean)^2`, then average those values. Add the variance and standard deviation (the square root of the variance — you may use `Math.sqrt` from the `Math` module) to the printed output.

Chapter 15 — Recursion

In the previous two chapters we saw how procedures can call other procedures, and how this lets us build programs from small, well-named pieces. There is one further possibility we have not yet considered: a procedure calling *itself*. This is called **recursion**, and it is one of the most elegant and powerful ideas in all of programming.

Recursion is not merely a curiosity. Some problems have a naturally recursive structure — they are defined in terms of smaller versions of themselves — and for these problems a recursive solution is often cleaner, shorter, and easier to understand than the equivalent iterative solution. Other problems are better solved with loops. Knowing when to use each approach is an important part of becoming a skilled programmer, and this chapter will help you develop that judgement.

15.1 The Basic Idea

Let us start with the simplest possible example. Suppose we want to count down from some number to 1, printing each value. We could do this with a loop, but here is how it looks as a recursive procedure:

```
PROCEDURE CountDown(n : INTEGER);
BEGIN
  IF n > 0 THEN
    Out.Int(n, 0);
    Out.Ln;
    CountDown(n - 1)    (* the procedure calls itself *)
  END
END CountDown;
```

When `CountDown(3)` is called:

- It prints 3, then calls `CountDown(2)`.
- That call prints 2, then calls `CountDown(1)`.
- That call prints 1, then calls `CountDown(0)`.
- That call finds `n > 0` is `FALSE`, does nothing, and returns.
- Control returns to `CountDown(1)`, which also returns.
- Control returns to `CountDown(2)`, which also returns.
- Control returns to `CountDown(3)`, which also returns.

Output:

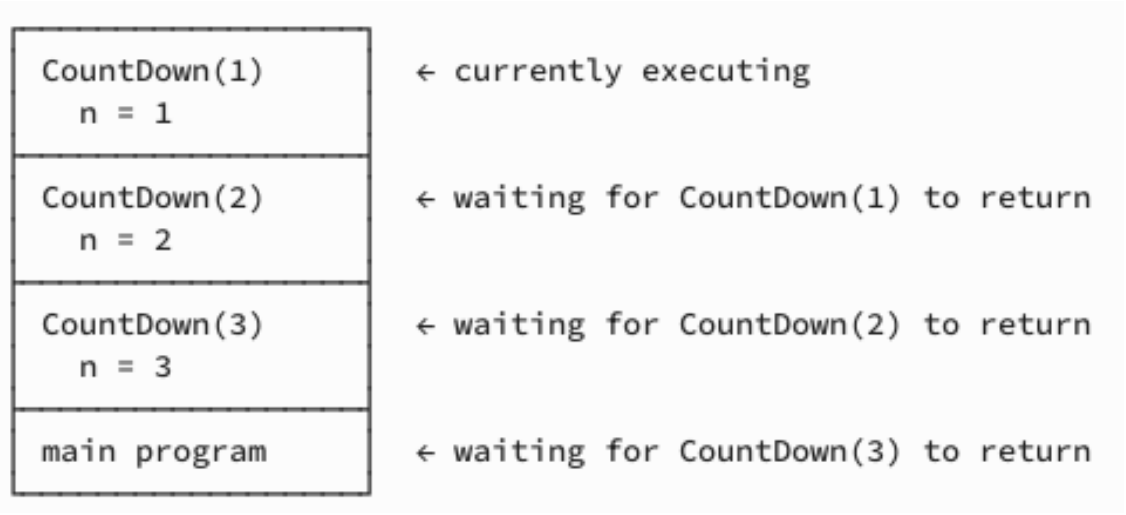
```
3
2
1
```

The procedure calls itself with a *smaller* argument each time — `n - 1` instead of `n`. This is what makes it terminate: eventually `n` reaches `0` and the condition `n > 0` is false, so no further call is made.

15.2 The Call Stack

To understand recursion, it helps to understand what happens in memory when a procedure calls itself. Each call to a procedure creates a fresh set of local variables and parameters — a **stack frame** — in an area of memory called the **call stack**. When the procedure returns, its stack frame is removed.

When `CountDown(3)` calls `CountDown(2)`, the stack looks like this:



Each call has its own copy of `n`. They are completely independent — changing `n` in one call does not affect the `n` in any other call.

When `CountDown(1)` calls `CountDown(0)`, a fourth frame is added. When `CountDown(0)` returns, its frame is removed, and execution continues in `CountDown(1)`. The stack unwinds one frame at a time until control returns to the main program.

The call stack has a finite size. If a recursive procedure calls itself too many times without returning, the stack fills up and the program traps with a **stack overflow**. This is why every recursive procedure must have a condition that stops the recursion — a **base case** — as we discuss next.

15.3 Base Cases and Recursive Cases

Every correct recursive procedure has two parts:

A **base case** — a condition under which the procedure does not call itself, but instead returns directly (or does some simple action). The base case is what stops the recursion. Without it, the procedure would call itself forever.

A **recursive case** — a call to the procedure itself, always with arguments that are closer to the base case. Each recursive call must make progress towards the base case. If it does not, the recursion will never terminate.

In `CountDown`, the base case is `n = 0` (or more precisely, `n <= 0`) — when this is true, nothing happens and the procedure returns. The recursive case is `CountDown(n - 1)`, which reduces `n` by 1, making progress towards 0.

Before writing any recursive procedure, ask yourself: - What is the base case? When should the recursion stop? - What is the recursive case? How does each call get closer to the base case?

If you cannot answer both questions, you are not ready to write the code.

15.4 Factorial

The **factorial** of a non-negative integer n, written n!, is the product of all integers from 1 to n. By definition, 0! = 1. The first few values are:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Factorial has a natural recursive definition: n! = n × (n-1)! for n > 0. In other words, the factorial of n is n times the factorial of n-1. This maps directly onto a recursive function procedure:

```
PROCEDURE Factorial(n : INTEGER) : INTEGER;
VAR
  result : INTEGER;
BEGIN
  IF n <= 0 THEN
    result := 1
  ELSE
    result := n * Factorial(n - 1)
  END;
  RETURN result
END Factorial;
```

The base case is `n <= 0`, which returns 1. The recursive case multiplies `n` by `Factorial(n - 1)`. Let us trace `Factorial(4)`:

```
Factorial(4)
  = 4 * Factorial(3)
  = 4 * (3 * Factorial(2))
  = 4 * (3 * (2 * Factorial(1)))
  = 4 * (3 * (2 * (1 * Factorial(0))))
  = 4 * (3 * (2 * (1 * 1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

Here is the complete factorial program:

Example 15.1

```
MODULE FactDemo;
  IMPORT In, Out;
VAR
  n : INTEGER;

PROCEDURE Factorial(n : INTEGER) : INTEGER;
VAR
  result : INTEGER;
BEGIN
  IF n <= 0 THEN
    result := 1
  ELSE
    result := n * Factorial(n - 1)
  END;
  RETURN result
END Factorial;

BEGIN
  Out.String("Enter a non-negative integer: ");
  Out.Ln;
  In.Int(n);
  IF In.Done & (n >= 0) THEN
    Out.Int(n, 0);
    Out.String("! = ");
    Out.Int(Factorial(n), 0);
    Out.Ln
  ELSE
    Out.String("Please enter a non-negative integer.");
    Out.Ln
  END
END FactDemo.
```

With input 5:

```
5! = 120
```

Note that factorial grows very rapidly. `12!` is 479,001,600, which is still within the range of a 32-bit integer, but `13!` overflows it. With OBNC's 64-bit integers you can go further, but even so, be cautious with large inputs.

15.5 Fibonacci

The **Fibonacci sequence** is another classic example of recursive definition. Each number in the sequence is the sum of the two preceding ones:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

Formally: `Fib(0) = 0`, `Fib(1) = 1`, and `Fib(n) = Fib(n-1) + Fib(n-2)` for n > 1. This has two base cases (n = 0 and n = 1) and one recursive case with two recursive calls:

```
PROCEDURE Fib(n : INTEGER) : INTEGER;
VAR
  result : INTEGER;
BEGIN
  IF n <= 0 THEN
    result := 0
  ELSIF n = 1 THEN
    result := 1
  ELSE
    result := Fib(n - 1) + Fib(n - 2)
  END;
  RETURN result
END Fib;
```

Here is a complete program that prints the first 12 Fibonacci numbers:

Example 15.2

```
MODULE FibDemo;
  IMPORT Out;
VAR
  i : INTEGER;

PROCEDURE Fib(n : INTEGER) : INTEGER;
VAR
  result : INTEGER;
BEGIN
  IF n <= 0 THEN
    result := 0
  ELSIF n = 1 THEN
    result := 1
  ELSE
    result := Fib(n - 1) + Fib(n - 2)
  END;
  RETURN result
END Fib;

BEGIN
  FOR i := 0 TO 11 DO
    Out.String("Fib(");
```

```
    Out.Int(i, 0);
    Out.String(") = ");
    Out.Int(Fib(i), 0);
    Out.Ln
  END
END FibDemo.
```

Output:

```
Fib(0)  = 0
Fib(1)  = 1
Fib(2)  = 1
Fib(3)  = 2
Fib(4)  = 3
Fib(5)  = 5
Fib(6)  = 8
Fib(7)  = 13
Fib(8)  = 21
Fib(9)  = 34
Fib(10) = 55
Fib(11) = 89
```

A Warning About Naive Fibonacci

The recursive Fibonacci procedure is a clean illustration of recursion with two base cases, but it is a notoriously poor implementation for large values of `n`. The problem is that it recalculates the same values over and over. `Fib(5)` calls `Fib(4)` and `Fib(3)`; `Fib(4)` calls `Fib(3)` and `Fib(2)`; `Fib(3)` is computed twice, `Fib(2)` three times, and so on. The number of calls grows exponentially with `n` — computing `Fib(40)` requires over a billion calls.

For large Fibonacci values, an iterative solution using a loop is far more efficient:

```
PROCEDURE FibIter(n : INTEGER) : INTEGER;
VAR
  a, b, i, temp : INTEGER;
BEGIN
  IF n <= 0 THEN
    a := 0
  ELSIF n = 1 THEN
    a := 1
  ELSE
    a := 0; b := 1;
    FOR i := 2 TO n DO
      temp := a + b;
      a := b;
```

```
      b := temp
    END;
    a := b
  END;
  RETURN a
END FibIter;
```

This is a good illustration of the broader point: recursion is elegant when the problem structure naturally calls for it, but it is not always the most efficient approach.

15.6 When to Use Recursion vs Loops

Recursion and iteration (loops) are equally powerful in theory — anything you can do with one, you can do with the other. The choice between them is a matter of clarity and efficiency.

Use recursion when: - The problem has a naturally recursive definition (like factorial, Fibonacci, or traversing a tree structure). - The recursive solution is significantly clearer than the iterative one. - The depth of recursion is modest (not thousands of levels deep).

Use iteration when: - The problem is naturally sequential — process each item one after another. - Efficiency matters and the recursion would repeat work (like naive Fibonacci). - The recursion depth could be large, risking stack overflow.

A useful heuristic: if you find yourself writing a loop that naturally decomposes into "handle this item, then do the same thing for the rest", recursion is probably a good fit. If the problem is naturally a scan or accumulation over a sequence, a loop is almost certainly cleaner.

15.7 Binary Search

Binary search is an algorithm for finding a value in a **sorted** array. Rather than scanning from the beginning (which takes time proportional to the array size), binary search repeatedly halves the search range: compare the target with the middle element, and discard the half that cannot contain the target. This is a much more efficient way of searching for an item than a linear search.

The algorithm is naturally recursive. The search on a range is defined in terms of the same search on a smaller range.

How it Works

Suppose we have the sorted array:

```
Index:  0   1   2   3   4   5   6   7   8   9
Value: 10  20  30  40  50  60  70  80  90 100
```

and we are searching for the value 40. A linear scan would start at index 0 and check each element in turn. Binary search is smarter: it starts in the middle.

Step 1. The middle index is `(0 + 9) DIV 2 = 4`. The value there is `50`. Since `40 < 50`, the target must be in the left half — indices `0` to `3`. We discard the right half entirely.

Step 2. The middle of the remaining range is `(0 + 3) DIV 2 = 1`. The value there is `20`. Since `40 > 20`, the target must be in the right portion of what remains — indices `2` to `3`.

Step 3. The middle is `(2 + 3) DIV 2 = 2`. The value there is `30`. Since `40 > 30`, we narrow further to indices `3` to `3`.

Step 4. The middle is `(3 + 3) DIV 2 = 3`. The value there is `40`. Found — the target is at index `3`.

Four steps to find a value in an array of ten elements. If the target had not been present — say we were searching for `45` — step 4 would have left us with an empty range (low > high), and we would know immediately that `45` is not in the array. The efficiency comes from the fact that each step eliminates half the remaining candidates, so even an array of a million elements requires at most twenty steps.

Developing the Algorithm

First refinement:

```
Search for target in the array.
```

Second refinement:

```
If the range is empty, the target is not present.
Otherwise, compare target with the middle element.
If equal, we have found it.
If target is smaller, search the left half.
If target is larger, search the right half.
```

Third refinement:

```
BinarySearch(a, target, low, high):
  If low > high: return -1  (* not found *)
  mid := (low + high) DIV 2
  If a[mid] = target: return mid
  If target < a[mid]: return BinarySearch(a, target, low, mid - 1)
  Else:               return BinarySearch(a, target, mid + 1, high)
```

The base case is `low > high` — an empty range means the target is not present. Each recursive call reduces the range by approximately half, so the algorithm terminates quickly even for large arrays.

The Program

```
MODULE BinSearch;
  IMPORT In, Out;
CONST
  N = 10;
VAR
```

```
  data   : ARRAY N OF INTEGER;
  target : INTEGER;
  pos    : INTEGER;
  i      : INTEGER;

PROCEDURE BinarySearch(a : ARRAY OF INTEGER;
          target, low, high : INTEGER) : INTEGER;
VAR
  mid    : INTEGER;
  result : INTEGER;
BEGIN
  IF low > high THEN
    result := -1
  ELSE
    mid := (low + high) DIV 2;
    IF a[mid] = target THEN
      result := mid
    ELSIF target < a[mid] THEN
      result := BinarySearch(a, target, low, mid - 1)
    ELSE
      result := BinarySearch(a, target, mid + 1, high)
    END
  END;
  RETURN result
END BinarySearch;

BEGIN
  (* Fill with sorted data *)
  FOR i := 0 TO N - 1 DO
    data[i] := (i + 1) * 10
  END;
  Out.String("Array: ");
  FOR i := 0 TO N - 1 DO
    Out.Int(data[i], 5)
  END;
  Out.Ln;
  Out.String("Enter a value to search for: ");
  Out.Ln;
  In.Int(target);
  IF In.Done THEN
    pos := BinarySearch(data, target, 0, N - 1);
    IF pos >= 0 THEN
      Out.String("Found at index ");
      Out.Int(pos, 0)
    ELSE
      Out.String("Not found")
```

```
    END;
    Out.Ln
  END
END BinSearch.
```

With input 40:

```
Array:    10   20   30   40   50   60   70   80   90  100
Found at index 3
```

With input 45:

```
Array:    10   20   30   40   50   60   70   80   90  100
Not found
```

Let us trace the search for target 40 in the array [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]:

```
BinarySearch(data, 40, 0, 9)
  mid = 4, data[4] = 50. 40 < 50, so search left:
  BinarySearch(data, 40, 0, 3)
    mid = 1, data[1] = 20. 40 > 20, so search right:
    BinarySearch(data, 40, 2, 3)
      mid = 2, data[2] = 30. 40 > 30, so search right:
      BinarySearch(data, 40, 3, 3)
        mid = 3, data[3] = 40. Found! Return 3.
```

Four calls to find the answer in an array of ten elements. For an array of a thousand elements, binary search would need at most ten calls. For a million elements, at most twenty. This dramatic efficiency — the number of steps grows as the logarithm of the array size — is why binary search is one of the most important algorithms in computing. The recursive structure makes this naturally clear: each call halves the problem size.

15.8 Recursion and Linked Lists

Recursion pairs naturally with linked lists, because a linked list is itself a recursively defined structure: a list is either empty (`NIL`), or it is a node followed by another list. This recursive definition suggests recursive algorithms.

Here is a recursive procedure that computes the length of a linked list — the number of nodes from a given pointer to the end:

```
TYPE
  NodePtr = POINTER TO Node;
  Node    = RECORD
    value : INTEGER;
    next  : NodePtr
  END;

PROCEDURE Length(p : NodePtr) : INTEGER;
VAR
  result : INTEGER;
BEGIN
  IF p = NIL THEN
    result := 0
  ELSE
    result := 1 + Length(p.next)
  END;
  RETURN result
END Length;
```

The base case is `p = NIL` — an empty list has length 0. The recursive case says: the length of a non-empty list is 1 (for the current node) plus the length of the rest of the list. This is a perfect match between the recursive structure of the problem and the recursive structure of the solution.

Here is a recursive procedure that prints all values in a linked list:

```
PROCEDURE PrintList(p : NodePtr);
BEGIN
  IF p # NIL THEN
    Out.Int(p.value, 0);
    Out.String(" ");
    PrintList(p.next)
  END
END PrintList;
```

And here is the same task but printing in **reverse** order — a task that is notably harder to do iteratively but falls out naturally from recursion:

```
PROCEDURE PrintReverse(p : NodePtr);
BEGIN
  IF p # NIL THEN
    PrintReverse(p.next);    (* recurse first *)
    Out.Int(p.value, 0);     (* print on the way back *)
    Out.String(" ")
  END
END PrintReverse;
```

By making the recursive call *before* printing, `PrintReverse` descends to the end of the list first, then prints values as the calls return — in reverse order. This elegant reversal of behaviour simply by reordering the recursive call and the print statement is a distinctly recursive pattern that has no simple iterative equivalent without an auxiliary stack.

Summary

- **Recursion** is a looping procedure characterised by calling itself. Each call has its own independent set of local variables and parameters on the **call stack**.
- Every recursive procedure must have a **base case** — a condition under which it returns without making a further recursive call — and a **recursive case** that calls itself with arguments closer to the base case.
- Without a base case, or if progress towards it is not made, the recursion will not terminate and will cause a **stack overflow**.
- **Factorial** and **Fibonacci** are classic recursive algorithms. The naive recursive Fibonacci is elegant but inefficient for large values — an iterative version is far faster.
- **Binary search** is an efficient recursive algorithm for finding a value in a sorted array. It halves the search range with each call, requiring at most $\log_2(n)$ calls for an array of `n` elements.
- **Linked lists** have a naturally recursive structure. Recursive procedures on linked lists are often more concise than iterative ones, and printing in reverse order is a natural application.
- Use recursion when the problem has natural recursive structure and the depth is modest. Use iteration when the problem is sequential or efficiency is critical.

Exercises

15.1 Write a recursive function procedure `SumTo(n : INTEGER) : INTEGER` that returns the sum of integers from 1 to `n`. For example, `SumTo(5)` should return 15. Write a complete program that tests it for several values. What is the base case?

15.2 Write a recursive function procedure `Power(base, exp : INTEGER) : INTEGER` that computes `base` raised to the power `exp` (where `exp >= 0`). The base case is `exp = 0` (any number to the power 0 is 1). Test it with several values and compare with OBNC's built-in power function when you reach Chapter 17.

15.3 Write a recursive procedure `PrintBinary(n : INTEGER)` that prints the binary representation of a positive integer. (Hint: the last binary digit is `n MOD 2`; the remaining digits are the binary representation of `n DIV 2`. Print the recursive call first, then the digit — why?)

15.4 Write a recursive function procedure `GCD(a, b : INTEGER) : INTEGER` that computes the greatest common divisor of two positive integers using Euclid's algorithm: `GCD(a, 0) = a` and `GCD(a, b) = GCD(b, a MOD b)`. Compare the recursive version with the iterative `WHILE/ELSIF` version from Chapter 6. Which do you find clearer?

15.5 Write a recursive function procedure `IsPalindrome(s : ARRAY OF CHAR; low, high : INTEGER) : BOOLEAN` that checks whether the characters of string `s` between indices `low` and `high` form a palindrome. The base cases are when `low >= high` (zero or one characters are always palindromes). The recursive case checks that `s[low] = s[high]` and that the inner portion is also a palindrome.

Write a program that tests it with several strings.

15.6 Extend the linked list example from Section 15.8 with a recursive function procedure `Sum(p : NodePtr) : INTEGER` that returns the sum of all values in the list. Build a list of five values, print the list using `PrintList`, then print the sum and the length using your recursive procedures.

15.7 Write a recursive procedure `Permute` that prints all permutations of the characters in a string. Start with `a : ARRAY OF CHAR` containing a short word (3 or 4 characters). The algorithm: if there is only one character left to permute, print the current arrangement; otherwise, for each position `i`, swap character `i` with the last character, recurse on the first `n-1` characters, then swap back. (This is a classic recursive algorithm — take your time with the base case and the swap-recurse-swap pattern before writing code.)

Chapter 16 — Modules

Throughout this book we have imported modules — `Out`, `In`, `Strings`, `Math` — and used the procedures and functions they provide. We have treated them as black boxes: we know what they do, but not how they are built or why they are organised the way they are. Now, having covered procedures, function procedures, and recursion, we have all the tools needed to understand modules fully — and to write our own.

This chapter explains what a module is, how identifiers are exported and imported, and how to design and write a module from scratch. The worked example walks through the creation of a `Sets` module that extends Oberon-07's built-in set operations with additional procedures not provided by the language itself.

16.1 What Is a Module?

A **module** is a named, self-contained unit of Oberon-07 source code. It contains declarations — constants, types, variables, and procedures — and an optional initialisation sequence that runs when the module is first loaded. Every Oberon-07 program we have written so far has been a module: each began with `MODULE name;` and ended with `END name.`

But modules serve a second purpose beyond housing a program's main logic: they are the unit of **code reuse** and **abstraction** in Oberon-07. A module can be written once, compiled once, and then imported by any number of other modules. The `Out`, `In`, and `Strings` modules you have been using are exactly this — compiled modules that expose a clean interface and hide their implementation details.

A module that imports and uses another module is called a **client** of that module — just as a customer is a client of a business that provides a service. The importing module depends on the exported interface of the module it imports, but need not know anything about how that interface is implemented.

This separation between **interface** (what a module provides) and **implementation** (how it provides it) is one of the most important ideas in software engineering. A client module — one that imports another — only needs to know the interface. It does not need to know, and cannot see, the private internals.

16.2 The IMPORT List

As you know, a module declares which other modules it uses in its `IMPORT` list, immediately after the module name:

```
MODULE MyProgram;
  IMPORT Out, In, Strings;
```

Each name in the list refers to a compiled module that must be available to the compiler. When `MyProgram` is compiled, the compiler reads the interfaces of `Out`, `In`, and `Strings` to check that they are used correctly.

You can give an imported module a local alias using `:=`:

```
IMPORT S := Strings;
```

After this, `S.Length` refers to `Strings.Length`. Aliases are useful when a module name is long or when two imported modules have similar names. In practice, aliases are rarely needed for the standard library modules.

16.3 Exporting Identifiers with *

By default, everything declared in a module is **private** — visible only within the module itself. To make an identifier visible to importing modules, you mark it with an **export marker**: an asterisk `*` immediately after the identifier name in its declaration.

```
MODULE Geometry;
CONST
  Pi* = 3.14159265;    (* exported — clients can use Geometry.Pi *)
TYPE
  Point* = RECORD      (* exported — clients can declare Point variables *)
    x, y : REAL        (* fields are NOT exported — private to the module *)
  END;
VAR
  count* : INTEGER;    (* exported read-only — clients can read but not write *)
  secret : INTEGER;    (* NOT exported — private to the module *)

PROCEDURE Area*(r : REAL) : REAL;   (* exported — clients can call Geometry.Area *)
BEGIN
  RETURN Pi * r * r
END Area;

PROCEDURE Helper;    (* NOT exported — private helper, invisible to clients *)
BEGIN
  INC(secret)
END Helper;

END Geometry.
```

A client module imports `Geometry` and can then use `Geometry.Pi`, `Geometry.Area`, and read `Geometry.count` — but it cannot write to `Geometry.count`, cannot call `Geometry.Helper`, and cannot access `secret` at all. The private details are completely hidden.

This is the key principle: **export the interface, hide the implementation**. Clients depend only on what is exported. If you later change how `Area` is calculated internally — switching to a different formula, adding caching, whatever — client modules are unaffected as long as the exported interface stays the same.

What Can Be Exported

Constants, types, variables, and procedures can all be exported. There is one important restriction on

variables: **exported variables are read-only in client modules**. A client can read the value of an exported variable but cannot assign to it. Only the module that owns the variable can modify it. This protects the module's internal state from being corrupted by external code.

Record fields cannot be individually exported — either the record type is exported (with `*` on the type name) or it is not. If a record type is exported, its fields are accessible to clients through the dot selector and all its fields are accessible to clients by name — individual fields cannot be selectively exported or hidden. The only way to keep record fields private is to not export the record type at all, in which case clients cannot declare variables of that type directly and must interact with the record solely through the procedures the module provides.

16.4 Compiling and Using a Module

A module is saved in a file whose name matches the module name, with the .obn extension. To use the Sets module in a program, save Sets.obn in the same folder as the client program. When the client is compiled, OBNC will automatically find and compile any modules it imports. So suppose the client module is named `TestSets.obn`, simply compile as normal:

```
obnc TestSets.obn
```

This compiles both TestSets and Sets in a single step, producing an executable that can be run with:

```
./TestSets
```

There is no need to compile `Sets.obn` separately first — OBNC handles the dependencies automatically as long as all the source files are in the same folder.

When OBNC compiles a module it creates a hidden folder called `.obnc` in the same directory as the source files. This folder contains the intermediate C files, compiled object files, and interface information that `OBNC` generates during the build process. You will rarely need to look inside it, but it is worth knowing it is there. If you ever encounter puzzling compilation errors after making structural changes to a module — renaming an exported procedure, for example — deleting the `.obnc` folder and recompiling from scratch will often resolve them, as it forces OBNC to regenerate everything from the current source files.

The obncdoc Tool

OBNC provides a tool called `obncdoc` that extracts the public interface of a compiled module and displays it as a `DEFINITION` block — a summary of everything exported, without any implementation details. For example, running `obncdoc Sets` after compiling `Sets.obn` would show something like:

```
DEFINITION Sets;
  PROCEDURE print(s: SET);
  PROCEDURE card(s: SET): INTEGER;
  PROCEDURE subset(s1, s2: SET): BOOLEAN;
  PROCEDURE properSubset(s1, s2: SET): BOOLEAN;
  PROCEDURE disjoint(s1, s2: SET): BOOLEAN;
```

```
END Sets.
```

This is the interface a client module sees. The implementation — the loops, the comparisons, the local variables — is invisible. `obncdoc` is a useful tool for checking what a module actually exports and for documenting your own modules.

16.5 Scope and Visibility in Modules

The scope rules for modules follow naturally from what we have already seen:

A **module-level** identifier (declared in the module's `CONST`, `TYPE`, or `VAR` section, or as a procedure) is visible throughout the entire module. If it carries an export marker `*`, it is also visible in any module that imports it.

A **procedure-level** identifier (local to a procedure) is visible only within that procedure. It is never exported, regardless of whether the containing module exports other things.

An imported identifier is accessed using the qualified form: `ModuleName.Name`. You cannot use just `Name` to refer to something from an imported module — the module prefix is always required. This prevents name clashes between modules: two different modules can both export a procedure called `Print` without conflict, because they are accessed as `ModuleA.Print` and `ModuleB.Print`.

16.6 The Module Initialisation Block

A module may include a `BEGIN`…`END` block — the **initialisation block** — that runs automatically when the module is first imported. This is used to set up the module's internal state before any of its procedures are called.

For example, a module that uses a counter might initialise it to zero:

```
MODULE Counter;
VAR
  count* : INTEGER;

PROCEDURE Increment*;
BEGIN
  INC(count)
END Increment;

PROCEDURE Reset*;
BEGIN
  count := 0
END Reset;

BEGIN
  count := 0    (* initialisation: runs when the module is first imported *)
END Counter.
```

When any program imports `Counter`, `count` is guaranteed to be `0` before any procedure is called. This is important: without the initialisation block, `count` would have an undefined initial value.

If a module has no state to initialise, the `BEGIN`…`END` block can be omitted entirely. The `Sets` module we will write shortly has no module-level variables and therefore needs no initialisation block.

16.7 Example: The Sets Module

In Chapter 12 we introduced Oberon-07's built-in `SET` type and its operators — union, intersection, difference, and so on. The language provides the basic operations, but several useful set-theoretic concepts are missing: the **cardinality** of a set (the number of elements it contains), the **subset** relationship, the **proper subset** relationship, and the **disjoint** test (whether two sets share no elements).

These are natural candidates for a module: they are general-purpose utilities that any program working with sets might want, they are independent of each other, and they have clear, well-defined specifications. We will build them into a `Sets` module, walking through the design of each procedure.

Designing the Interface

Before writing any code, it is good practice to decide what the module should export — its interface. One useful way to record this decision is as a `DEFINITION` block: a human-readable summary of the exported identifiers and their types, written as a design document. Note that `OBNC` does not *require* a `DEFINITION` file in order to compile a module — the compiler generates everything it needs from the `.obn` source file. The `DEFINITION` format is simply a documentation convention, useful for thinking clearly about the interface before committing to an implementation, and it can also be generated automatically from a compiled module using the obncdoc tool. Here is a `DEFINITION` block for the `Sets` module:

```
DEFINITION Sets;
  PROCEDURE print(s: SET);
    (* prints the elements of s, one per line *)
  PROCEDURE card(s: SET): INTEGER;
    (* returns the number of elements in s *)
  PROCEDURE subset(s1, s2: SET): BOOLEAN;
    (* returns TRUE if s1 is a subset of s2 *)
  PROCEDURE properSubset(s1, s2: SET): BOOLEAN;
    (* returns TRUE if s1 is a proper subset of s2 *)
  PROCEDURE disjoint(s1, s2: SET): BOOLEAN;
    (* returns TRUE if s1 and s2 have no elements in common *)
END Sets.
```

Five procedures, each doing one clearly defined thing. Notice that there are no exported variables or types — this module is purely a collection of utility procedures.

Implementing the Procedures

`print` needs to scan all possible set elements (0 to 63 in OBNC on a 64-bit system) and print those that are present. We use the `IN` operator to test membership and a `FOR` loop to cover the full range:

```
PROCEDURE print*(s : SET);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO 63 DO
    IF i IN s THEN
      Out.Int(i, 3);
      Out.Ln
    END
  END;
  Out.Ln
END print;
```

Each member is printed right-justified in a field of width 3, followed by a newline. A final `Out.Ln` adds a blank line after the set, separating it visually from subsequent output.

`card` (cardinality) counts the elements. The approach is the same: scan each possible element and count those present:

```
PROCEDURE card*(s : SET) : INTEGER;
VAR
  i, c : INTEGER;
BEGIN
  c := 0;
  FOR i := 0 TO 63 DO
    IF i IN s THEN
      INC(c)
    END
  END;
  RETURN c
END card;
```

`subset` tests whether every element of `s1` is also in `s2`. The mathematical definition is: `s1` is a subset of `s2` if and only if `s1 - s2` is empty. The difference `s1 - s2` contains elements that are in `s1` but not in `s2`. If that is empty, there are no elements in `s1` that are missing from `s2`:

```
PROCEDURE subset*(s1, s2 : SET) : BOOLEAN;
BEGIN
  RETURN s1 - s2 = {}
END subset;
```

This is a beautifully concise implementation — a single expression captures the mathematical definition exactly.

properSubset is a subset that is not equal to the whole set. `s1` is a proper subset of `s2` if `s1` is a subset of `s2` and `s1 # s2` (i.e. there are elements in `s2` that are not in `s1`):

```
PROCEDURE properSubset*(s1, s2 : SET) : BOOLEAN;
BEGIN
  RETURN (s1 - s2 = {}) & (s1 # s2)
END properSubset;
```

disjoint tests whether two sets share no elements — their intersection is empty:

```
PROCEDURE disjoint*(s1, s2 : SET) : BOOLEAN;
BEGIN
  RETURN s1 * s2 = {}
END disjoint;
```

Again, a single expression captures the definition precisely.

The Complete Module

Putting it all together, and following Oberon-07's convention that the module name matches the filename:

```
MODULE Sets;
  IMPORT Out;

PROCEDURE print*(s : SET);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 TO 63 DO
    IF i IN s THEN
      Out.Int(i, 3);
      Out.Ln
    END
  END;
  Out.Ln
END print;

PROCEDURE card*(s : SET) : INTEGER;
VAR
  i, c : INTEGER;
BEGIN
  c := 0;
```

```
  FOR i := 0 TO 63 DO
    IF i IN s THEN
      INC(c)
    END
  END;
  RETURN c
END card;

PROCEDURE subset*(s1, s2 : SET) : BOOLEAN;
BEGIN
  RETURN s1 - s2 = {}
END subset;

PROCEDURE properSubset*(s1, s2 : SET) : BOOLEAN;
BEGIN
  RETURN (s1 - s2 = {}) & (s1 # s2)
END properSubset;

PROCEDURE disjoint*(s1, s2 : SET) : BOOLEAN;
BEGIN
  RETURN s1 * s2 = {}
END disjoint;

END Sets.
```

There is no `BEGIN...END` initialisation block because the module has no state to initialise — it contains only procedures, no module-level variables.

Testing the Module

A good module should be accompanied by a test program. Here is a driver that exercises all five procedures with known inputs and prints the results:

```
MODULE TestSets;
  IMPORT Out, Sets;
VAR
  a, b : SET;
BEGIN
  a := {1, 2, 3, 4, 5};
  b := {3, 4, 5, 6, 7, 8};
  Out.String("Set a:"); Out.Ln;
  Sets.print(a);
  Out.String("Set b:"); Out.Ln;
  Sets.print(b);
  Out.String("Cardinality of a: ");
  Out.Int(Sets.card(a), 0); Out.Ln;
```

```
  Out.String("Cardinality of b: ");
  Out.Int(Sets.card(b), 0); Out.Ln;
  Out.String("{1,2} subset of a?       ");
  IF Sets.subset({1, 2}, a) THEN
    Out.String("Yes")
  ELSE
    Out.String("No")
  END; Out.Ln;
  Out.String("a subset of {1..10}?     ");
  IF Sets.subset(a, {1..10}) THEN
    Out.String("Yes")
  ELSE
    Out.String("No")
  END; Out.Ln;
  Out.String("{1,2} proper subset of a?");
  IF Sets.properSubset({1, 2}, a) THEN
    Out.String("Yes")
  ELSE
    Out.String("No")
  END; Out.Ln;
  Out.String("a proper subset of a?    ");
  IF Sets.properSubset(a, a) THEN
    Out.String("Yes")
  ELSE
    Out.String("No")
  END; Out.Ln;
  Out.String("a and b disjoint?        ");
  IF Sets.disjoint(a, b) THEN
    Out.String("Yes")
  ELSE
    Out.String("No")
  END; Out.Ln;
  Out.String("{1,2} and {6,7} disjoint?");
  IF Sets.disjoint({1, 2}, {6, 7}) THEN
    Out.String("Yes")
  ELSE
    Out.String("No")
  END; Out.Ln
END TestSets.
```

Output:

```
Set a:
  1
  2
```

```
  3
  4
  5

Set b:
  3
  4
  5
  6
  7

Cardinality of a: 5
Cardinality of b: 5
{1,2} subset of a?       Yes
a subset of {1..10}?     Yes
{1,2} proper subset of a?Yes
a proper subset of a?    No
a and b disjoint?        No
{1,2} and {6,7} disjoint?Yes
```

Every procedure is exercised with at least one positive and one negative case. `properSubset` is tested with a genuinely proper subset and with a set compared to itself (which should return `FALSE`). `disjoint` is tested with overlapping and non-overlapping sets.

This test-alongside-implementation habit — writing a driver to verify your module before using it in a larger program — is exactly the approach advocated in Chapter 13 for individual procedures. For modules, the same principle applies at a larger scale.

A Note on Module Design

The `Sets` module is a good example of a well-designed module: it has a narrow, coherent purpose (extending set operations), a clean interface (five clearly named procedures), no exported state (no global variables), and implementations that are as simple as possible. Each procedure does one thing and does it correctly.

As your programs grow larger and are spread across multiple modules, these qualities — coherence, simplicity, minimal exported state — become increasingly important. A module that tries to do too many things, or that exposes too much of its internal state, becomes difficult to maintain and difficult for clients to use correctly.

Summary

- A **module** is a named, self-contained compilation unit containing declarations and an optional initialisation sequence. It is the unit of code reuse in Oberon-07.
- Identifiers are exported by marking them with `*` in their declaration. Exported identifiers are accessed in client modules as `ModuleName.Identifier`.
- **Exported variables are read-only** in client modules — only the owning module can modify them.

- The `IMPORT` list specifies which modules a module depends on. An alias can be given with `:=`.
- The **initialisation block** (`BEGIN...END`) runs automatically when the module is first imported. It is used to set up module-level state. If there is no state to initialise, the block can be omitted.
- The principle of **interface vs implementation**: export only what clients need; hide the rest. A well-designed module can change its implementation without affecting any client.
- The `obncdoc` tool extracts and displays a module's exported interface as a `DEFINITION` block.
- A module file must be named after the module it contains, with the `.obn` extension, and compiled before any module that imports it.

Exercises

16.1 Compile the `Sets` module and `TestSets` program from the worked example. Run `TestSets` and verify the output matches what is shown. Then run `obncdoc Sets` and examine the interface it produces.

16.2 Add a procedure `union*(s1, s2 : SET) : SET` to the `Sets` module that returns the union of two sets, and a procedure `intersection*(s1, s2 : SET) : SET` that returns their intersection. Update `TestSets` to test both new procedures. (Note: returning a `SET` from a function procedure is valid since `SET` is a basic type.)

16.3 Write a module `Maths` that exports the following function procedures: `Min(a, b : INTEGER) : INTEGER`, `Max(a, b : INTEGER) : INTEGER`, and `Clamp(x, lo, hi : INTEGER) : INTEGER` (from Chapter 14's exercises). Write a test program that verifies each procedure with several inputs.

16.4 Write a module `Stats` that exports procedures for computing the sum, mean, minimum, and maximum of an array of reals, drawing on the `MathUtils` program from Chapter 14. The exported procedures should take open array parameters. Write a test program that reads a list of numbers and prints all four statistics.

16.5 Write a module `StrUtils` that exports the following procedures: `IsAllUpper(s : ARRAY OF CHAR) : BOOLEAN` (returns `TRUE` if every letter in `s` is uppercase), `IsAllDigits(s : ARRAY OF CHAR) : BOOLEAN` (returns `TRUE` if every character is a digit), and `CountChar(s : ARRAY OF CHAR; ch : CHAR) : INTEGER` (returns the number of occurrences of `ch` in `s`). Write a test program.

16.6 Write a module `IntStack` that implements a simple stack of integers using an array and a count variable (as described in the Landström article excerpt in the language reference material). Export procedures `Push(x : INTEGER)`, `Pop(VAR x : INTEGER)`, `Clear`, and an exported read-only variable `count`. Write a test program that pushes several values, pops them, and verifies the results using `ASSERT`.

Part IV — The OBNC Standard Libraries

- Chapter 17 — More Input & Output
- Chapter 18 — The Maths Module
- Chapter 19 — The Files Module
- Chapter 20 — Error Handling & Assertions
- Chapter 21 — Miscellaneous Topics

Chapter 17 — More Input & Output

Part IV of this book surveys the standard libraries that OBNC provides beyond the core `In`, `Out`, and `Strings` modules we have already used. We begin with two practical topics that arise constantly in real programs: formatting numbers for output, and reading arguments passed to a program on the command line.

We also revisit a limitation that was flagged back in Chapters 3 and 4: the `Out.Real` procedure displays real numbers in scientific notation, which is often not what you want. This chapter introduces a small module — Dec — that solves this problem cleanly.

17.1 Decimal Output for Real Numbers

When you call `Out.Real(x, 0)`, OBNC displays the value in scientific notation:

```
3.14159265E+00
1.23456789E+02
-7.50000000E-01
```

For many purposes this is inconvenient. A price of `9.99` should not appear as `9.99000E+00`, and a temperature of `36.6` should not appear as `3.66000E+01`. What we want is plain decimal notation — `9.99`, `36.6` — with control over the number of decimal places shown.

OBNC does not provide a built-in procedure for this, but the `Dec` module (available from the download link given in the preface) fills the gap. It exports a single procedure:

```
PROCEDURE print(x : REAL; w, dp : BYTE);
```

- `x` is the real number to print.
- `w` is the total field width (the minimum number of characters to occupy, right-justified with leading spaces if needed). Pass `0` for no padding.
- `dp` is the number of decimal places to show.

The module handles rounding, negative numbers, zero decimal places, and field width padding automatically. You do not need to understand how it works — simply place `Dec.obn` in the same folder as your program and import it.

Example 17.1: Using Dec.print

```
MODULE RealDemo;
  IMPORT Out, Dec;
VAR
  price  : REAL;
  temp   : REAL;
  pi     : REAL;
BEGIN
```

```
  price := 9.99;
  temp  := 36.6;
  pi    := 3.14159265;
  Out.String("Price:       "); Dec.print(price, 0, 2); Out.Ln;
  Out.String("Temperature: "); Dec.print(temp,  0, 1); Out.Ln;
  Out.String("Pi:          "); Dec.print(pi,    0, 5); Out.Ln;
  Out.Ln;
  (* Field width example -- right-justified in 10 columns *)
  Out.String("In a table:"); Out.Ln;
  Dec.print(1.5,    10, 2); Out.Ln;
  Dec.print(99.99,  10, 2); Out.Ln;
  Dec.print(-7.25,  10, 2); Out.Ln;
  Dec.print(1000.0, 10, 2); Out.Ln
END RealDemo.
```

Output:

```
Price:       9.99
Temperature: 36.6
Pi:          3.14159

In a table:
      1.50
     99.99
     -7.25
   1000.00
```

The first three calls use `w = 0` (no padding) and varying decimal places. The table example uses `w = 10` to right-justify each value in a ten-character column, exactly as `Out.Int` does for integers. Negative numbers are handled correctly — the minus sign is included in the field width.

Choosing Decimal Places

`Dec.print` rounds the value to the specified number of decimal places before printing. If `dp = 0`, the value is printed as a whole number with no decimal point:

```
Dec.print(3.7,  0, 0);   (* prints: 4   -- rounded  *)
Dec.print(3.14, 0, 0);   (* prints: 3   -- rounded  *)
Dec.print(3.14, 0, 1);   (* prints: 3.1            *)
Dec.print(3.14, 0, 2);   (* prints: 3.14           *)
Dec.print(3.14, 0, 4);   (* prints: 3.1400         *)
```

The maximum number of decimal places is 18, though the precision of OBNC's `REAL` type (64-bit IEEE 754 double precision) means that more than about 15 significant digits will not be meaningful.

17.2 The extConvert Module

The `extConvert` module provides four procedures for converting between numbers and their string representations. These are useful when you need to format numbers as strings — for building output lines, comparing numeric input, or passing numbers to procedures that expect strings.

```
IMPORT extConvert;
```

extConvert.IntToString

```
extConvert.IntToString(i : INTEGER; VAR s : ARRAY OF CHAR; VAR done : BOOLEAN)
```

Converts the integer `i` to its decimal string representation and stores it in `s`. If `s` is large enough, `done` is set to `TRUE`; if `s` is too small to hold the result, it is truncated and `done` is set to `FALSE`.

```
VAR
  s    : ARRAY 20 OF CHAR;
  ok   : BOOLEAN;
...
extConvert.IntToString(12345, s, ok);
IF ok THEN Out.String(s) END;    (* prints: 12345 *)
```

extConvert.RealToString

```
extConvert.RealToString(x : REAL; VAR s : ARRAY OF CHAR; VAR done : BOOLEAN)
```

Converts the real number `x` to a string representation (in the same scientific notation format as `Out.Real`) and stores it in `s`. `done` indicates success.

```
extConvert.RealToString(3.14, s, ok);
IF ok THEN Out.String(s) END;    (* prints something like: 3.14000E+00 *)
```

extConvert.StringToInt

```
extConvert.StringToInt(s : ARRAY OF CHAR; VAR i : INTEGER; VAR done : BOOLEAN)
```

Parses the string `s` as a decimal integer and stores the result in `i`. Leading whitespace is skipped. `done` is set to `TRUE` if the conversion succeeded, `FALSE` if the string does not contain a valid integer. Hexadecimal notation (`digit {hexDigit} "H"`) is also accepted.

```
extConvert.StringToInt("42", i, ok);
```

```
IF ok THEN Out.Int(i, 0) END;    (* prints: 42 *)

extConvert.StringToInt("hello", i, ok);
(* ok = FALSE: not a valid integer *)
```

extConvert.StringToReal

```
extConvert.StringToReal(s : ARRAY OF CHAR; VAR x : REAL; VAR done : BOOLEAN)
```

Parses the string `s` as a real number and stores the result in `x`. The expected format is `digit {digit} "." {digit}` optionally followed by a scale factor (`E` followed by an optional sign and digits). Leading whitespace is skipped. `done` indicates success.

```
extConvert.StringToReal("3.14", x, ok);
IF ok THEN Dec.print(x, 0, 2) END;    (* prints: 3.14 *)
```

Example 17.2: Validating Numeric String Input

A common use of `StringToInt` and `StringToReal` is to validate that a string entered by the user actually represents a number. This is sometimes more flexible than `In.Int`, which halts at the first non-digit character:

```
MODULE ValidateInput;
  IMPORT In, Out, extConvert;
CONST
  BufLen = 30;
TYPE
  Buf = ARRAY BufLen OF CHAR;
VAR
  s    : Buf;
  n    : INTEGER;
  ok   : BOOLEAN;
BEGIN
  Out.String("Enter an integer: ");
  In.Line(s);
  extConvert.StringToInt(s, n, ok);
  IF ok THEN
    Out.String("Valid integer: ");
    Out.Int(n, 0);
    Out.Ln
  ELSE
    Out.String("'");
    Out.String(s);
```

```
    Out.String("' is not a valid integer.");
    Out.Ln
  END
END ValidateInput.
```

With input `"42"`:

```
Valid integer: 42
```

With input `"hello"`:

```
'hello' is not a valid integer.
```

17.3 The extArgs Module

When you run an OBNC program from the command line, you can pass it additional information — **command-line arguments** — directly after the program name:

```
./myprogram firstarg secondarg thirdarg
```

The `extArgs` module gives you access to these arguments from within your program.

```
IMPORT extArgs;
```

The module exports one variable and one procedure:

extArgs.count

```
extArgs.count : INTEGER
```

An exported integer variable that holds the number of command-line arguments passed to the program. Arguments are numbered from 0; argument 0 is the first argument after the program name. If no arguments were given, `extArgs.count` is 0.

extArgs.Get

```
extArgs.Get(n : INTEGER; VAR arg : ARRAY OF CHAR; VAR res : INTEGER)
```

Retrieves the `n`-th command-line argument (0-based) and stores it in `arg`. If `arg` is not large enough to hold the argument, the result is truncated but always null-terminated, and `res` is set to the number of characters that were truncated. If the argument fits, `res` is 0.

Example 17.3: Reading Command-Line Arguments

```
MODULE ShowArgs;
  IMPORT Out, extArgs;
CONST
  ArgLen = 50;
VAR
  arg : ARRAY ArgLen OF CHAR;
  i   : INTEGER;
  res : INTEGER;
BEGIN
  Out.String("Number of arguments: ");
  Out.Int(extArgs.count, 0);
  Out.Ln;
  FOR i := 0 TO extArgs.count - 1 DO
    extArgs.Get(i, arg, res);
    Out.String("  arg[");
    Out.Int(i, 0);
    Out.String("] = ");
    Out.String(arg);
    Out.Ln
  END
END ShowArgs.
```

Running:

```
./ShowArgs hello world 42
```

Output:

```
Number of arguments: 3
  arg[0] = hello
  arg[1] = world
  arg[2] = 42
```

Arguments are always received as strings. If you need to treat an argument as a number, use `extConvert.StringToInt` or `extConvert.StringToReal` to convert it.

17.4 A Command-Line Unit Converter

Command-line arguments and numeric string conversion combine naturally in **command-line tools** — programs designed to be run from the terminal with their inputs given as arguments rather than typed interactively. This style is common in Unix/Linux environments and is well suited to small utility programs.

We will write a unit converter that converts a value from one unit to another. The program is called

as:

```
./Convert value from to
```

For example:

```
./Convert 100 km miles
./Convert 32 C F
./Convert 10 kg lbs
```

The program reads the three arguments, validates them, performs the conversion, and prints the result.

Design

We need to handle a set of known conversions. A natural representation is a table of records, each holding the source unit name, the target unit name, and the multiplication factor. We also need to handle the case where the arguments are missing, the value is not a valid number, or the unit pair is not recognised.

Example 17.4

```
MODULE Convert;
  IMPORT Out, extArgs, extConvert;
CONST
  ArgLen    = 20;
  NumConv   = 9;
TYPE
  ArgStr  = ARRAY ArgLen OF CHAR;
  ConvRec = RECORD
    from   : ArgStr;
    to     : ArgStr;
    factor : REAL
  END;
VAR
  table   : ARRAY NumConv OF ConvRec;
  fromStr : ArgStr;
  toStr   : ArgStr;
  valStr  : ArgStr;
  value   : REAL;
  result  : REAL;
  res     : INTEGER;
  ok      : BOOLEAN;
  i       : INTEGER;
  found   : BOOLEAN;
```

```
PROCEDURE InitTable;
BEGIN
  table[0].from := "km";     table[0].to := "miles";  table[0].factor := 0.621371;
  table[1].from := "miles";  table[1].to := "km";     table[1].factor := 1.60934;
  table[2].from := "kg";     table[2].to := "lbs";    table[2].factor := 2.20462;
  table[3].from := "lbs";    table[3].to := "kg";     table[3].factor := 0.453592;
  table[4].from := "m";      table[4].to := "ft";     table[4].factor := 3.28084;
  table[5].from := "ft";     table[5].to := "m";      table[5].factor := 0.3048;
  table[6].from := "C";      table[6].to := "F";      table[6].factor := 0.0;
  table[7].from := "F";      table[7].to := "C";      table[7].factor := 0.0;
  table[8].from := "l";      table[8].to := "gal";    table[8].factor := 0.219969
END InitTable;

PROCEDURE ConvertTemp(v : REAL; from : ARRAY OF CHAR) : REAL;
VAR
  r : REAL;
BEGIN
  IF from = "C" THEN
    r := v * 9.0 / 5.0 + 32.0
  ELSE
    r := (v - 32.0) * 5.0 / 9.0
  END;
  RETURN r
END ConvertTemp;

BEGIN
  InitTable;
  IF extArgs.count # 3 THEN
    Out.String("Usage: Convert <value> <from> <to>");
    Out.Ln;
    Out.String("Example: Convert 100 km miles");
    Out.Ln
  ELSE
    extArgs.Get(0, valStr, res);
    extArgs.Get(1, fromStr, res);
    extArgs.Get(2, toStr, res);
    extConvert.StringToReal(valStr, value, ok);
    IF ~ok THEN
      Out.String("Error: '");
      Out.String(valStr);
      Out.String("' is not a valid number.");
      Out.Ln
    ELSE
      found := FALSE;
      i     := 0;
      WHILE (i < NumConv) & ~found DO
```

```
        IF (table[i].from = fromStr) & (table[i].to = toStr) THEN
          found := TRUE;
          IF table[i].factor = 0.0 THEN
            result := ConvertTemp(value, fromStr)
          ELSE
            result := value * table[i].factor
          END
        END;
        INC(i)
      END;
      IF found THEN
        Out.String(valStr);
        Out.String(" ");
        Out.String(fromStr);
        Out.String(" = ");
        Out.Real(result, 0);
        Out.String(" ");
        Out.String(toStr);
        Out.Ln
      ELSE
        Out.String("Error: conversion from '");
        Out.String(fromStr);
        Out.String("' to '");
        Out.String(toStr);
        Out.String("' is not supported.");
        Out.Ln
      END
    END
  END
END Convert.
```

Sample runs:

```
./Convert 100 km miles
100 km = 6.21371E+01 miles

./Convert 32 C F
32 C = 8.96000E+01 F

./Convert 10 kg lbs
10 kg = 2.20462E+01 lbs

./Convert 50 km gal
Error: conversion from 'km' to 'gal' is not supported.

./Convert hello km miles
```

```
Error: 'hello' is not a valid number.

./Convert 100
Usage: Convert <value> <from> <to>
Example: Convert 100 km miles
```

You will notice that the output uses `Out.Real` — scientific notation — for the result. This is a good opportunity to replace those `Out.Real` calls with `Dec.print` now that you have it available. For example:

```
Dec.print(result, 0, 3);
```

would give more readable output:

```
100 km = 62.137 miles
```

A few design points are worth noting. The temperature conversions use a factor of `0.0` as a sentinel to indicate that a simple multiplication is not sufficient — they require the special `ConvertTemp` procedure instead. This is a pragmatic workaround given that the conversion table structure only holds a single factor. A more general design might use a procedure type for the conversion function, which would eliminate the special case entirely — an exercise left for the reader.

Summary

- The `Real` module provides `Dec.print(x, w, dp)` for printing real numbers in plain decimal notation with a specified field width and number of decimal places. Place `Real.obn` in the same folder as your program and import it as `Real`.
- `Dec.print` rounds to `dp` decimal places and right-justifies in a field of width `w`. Pass `w = 0` for no padding.
- The **`extConvert`** module provides conversions between numbers and strings: `IntToString`, `RealToString`, `StringToInt`, and `StringToReal`. All four use a `VAR done : BOOLEAN` parameter to indicate success or failure.
- `StringToInt` and `StringToReal` are useful for validating numeric input read as a string, and for parsing command-line arguments.
- The **`extArgs`** module gives access to command-line arguments. `extArgs.count` holds the number of arguments; `extArgs.Get(n, arg, res)` retrieves the `n`-th argument (0-based) as a string.
- Command-line arguments are always received as strings. Convert them to numbers using `extConvert` when needed.

Exercises

17.1 Write a program that prints a table of the squares and square roots of the integers from 1 to 15. Use `Dec.print` with 4 decimal places for the square roots. The `Math` module (Chapter 18) provides

`Math.sqrt` — you may use it here.

17.2 Write a program that reads a real number from the user and prints it with 0, 1, 2, 3, 4, and 5 decimal places, one per line, all right-justified in a field of width 12.

17.3 Write a program that reads a line of text from the user, attempts to convert it to an integer using `extConvert.StringToInt`, and prints either the value or an error message. Repeat until the user enters a valid integer.

17.4 Write a command-line program `WordCount` that accepts a word as its sole command-line argument and prints the number of characters it contains. If no argument is given, print a usage message. Run it as `./WordCount oberon`.

17.5 Extend the `Convert` program from Section 17.4 to use `Dec.print` instead of `Out.Real` for the result, with 3 decimal places. Test that the output is now in plain decimal notation.

17.6 Extend the `Convert` program to use a procedure type for the conversion function instead of the `factor = 0.0` sentinel. Declare a type `ConvFunc = PROCEDURE(x : REAL) : REAL` and store a procedure variable of this type in each table record. Write separate procedures for each conversion (including simple multiplication for non-temperature conversions) and assign them to the table entries in `InitTable`. This eliminates the special-case `IF` for temperature.

17.7 Write a command-line program `Stats` that accepts any number of real number arguments, computes their sum, mean, minimum, and maximum, and prints the results using `Dec.print`. Run it as `./Stats 3.0 7.5 2.0 9.5 4.0`. If no arguments are given, print a usage message.

Chapter 18 — The Maths Module

The arithmetic operators and predefined functions we have used so far — `+`, `-`, `*`, `/`, `DIV`, `MOD`, `ABS`, `FLT`, `FLOOR` — cover everyday calculation. But many programs need more: square roots, logarithms, trigonometric functions, exponentials. These are provided by OBNC's `Math` module, which implements the standard mathematical library specified in the Oakwood Guidelines for Oberon compiler developers.

To use any of the functions in this chapter, add `Math` to your import list:

```
IMPORT Math;
```

All `Math` functions take `REAL` arguments and return `REAL` results. Remember to use `Dec.print` from Chapter 17 if you want plain decimal output rather than scientific notation.

18.1 Constants

The `Math` module exports two mathematical constants: π and e (base of the natural logarithm):

```
Math.pi = 3.14159265358979  (* π, ratio of a circle's circumference to its diameter *)
Math.e  = 2.71828182845905  (* e, the base of the natural logarithm *)
```

These are available wherever a `REAL` constant is needed:

```
circumference := 2.0 * Math.pi * radius;
```

18.2 Square Root and Power

Math.sqrt

```
Math.sqrt(x : REAL) : REAL
```

Returns the square root of `x`. The argument must be non-negative; a negative value causes a runtime trap.

```
Dec.print(Math.sqrt(2.0),  0, 6);   Out.Ln;   (* 1.414214 *)
Dec.print(Math.sqrt(9.0),  0, 6);   Out.Ln;   (* 3.000000 *)
Dec.print(Math.sqrt(0.25), 0, 6);   Out.Ln;   (* 0.500000 *)
```

Math.power

```
Math.power(base, exp : REAL) : REAL
```

Returns base raised to the power exp. Both arguments are REAL, so fractional exponents are supported:

```
Dec.print(Math.power(2.0,  10.0), 0, 2);  Out.Ln;   (* 1024.00  *)
Dec.print(Math.power(2.0,  0.5),  0, 6);  Out.Ln;   (* 1.414214 -- same as sqrt(2) *)
Dec.print(Math.power(27.0, 1.0/3.0), 0, 6); Out.Ln; (* 3.000000 -- cube root of 27 *)
```

Math.round

```
Math.round(x : REAL) : REAL
```

Returns x rounded to the nearest integer value, returned as a REAL. If the fractional part is exactly 0.5, the value is rounded up. Note that the result is still a REAL — use FLOOR if you need an INTEGER.

```
Dec.print(Math.round(3.4), 0, 1);  Out.Ln;   (* 3.0 *)
Dec.print(Math.round(3.5), 0, 1);  Out.Ln;   (* 4.0 *)
Dec.print(Math.round(3.6), 0, 1);  Out.Ln;   (* 4.0 *)
Dec.print(Math.round(-2.5), 0, 1); Out.Ln;   (* -2.0 -- rounds up, towards zero *)
```

18.3 Exponential and Logarithmic Functions

Math.exp

```
Math.exp(x : REAL) : REAL
```

Returns e raised to the power x — the exponential function. This is the inverse of the natural logarithm:

```
Dec.print(Math.exp(0.0), 0, 6);  Out.Ln;   (* 1.000000 *)
Dec.print(Math.exp(1.0), 0, 6);  Out.Ln;   (* 2.718282 -- e itself *)
Dec.print(Math.exp(2.0), 0, 6);  Out.Ln;   (* 7.389056 *)
```

Math.ln

```
Math.ln(x : REAL) : REAL
```

Returns the natural logarithm of x (logarithm base e). The argument must be positive:

```
Dec.print(Math.ln(1.0),      0, 6);  Out.Ln;   (* 0.000000 -- ln(1) = 0  *)
Dec.print(Math.ln(Math.e),   0, 6);  Out.Ln;   (* 1.000000 -- ln(e) = 1  *)
Dec.print(Math.ln(Math.exp(3.0)), 0, 6); Out.Ln; (* 3.000000 -- inverse  *)
```

Math.log

```
Math.log(x, base : REAL) : REAL
```

Returns the logarithm of `x` with the specified `base`. For example, `Math.log(1000.0, 10.0)` returns the base-10 logarithm of 1000:

```
Dec.print(Math.log(1000.0, 10.0), 0, 6);  Out.Ln;   (* 3.000000 *)
Dec.print(Math.log(8.0,    2.0),  0, 6);  Out.Ln;   (* 3.000000 *)
Dec.print(Math.log(Math.e, Math.e), 0, 6); Out.Ln;  (* 1.000000 *)
```

The Relationship Between exp and ln

`Math.exp` and `Math.ln` are inverse functions; that is: `Math.ln(Math.exp(x)) = x` and `Math.exp(Math.ln(x)) = x` for any valid `x`. This relationship is sometimes useful for computing powers when both base and exponent are real: `base^exp = exp(exp * ln(base))`. This is exactly what `Math.power` computes internally.

18.4 Trigonometric Functions

All trigonometric functions in the `Math` module work in **radians**, not degrees. To convert degrees to radians, multiply by `Math.pi / 180.0`. To convert radians to degrees, multiply by `180.0 / Math.pi`.

```
degreesToRadians := Math.pi / 180.0;
radiansToDegrees := 180.0 / Math.pi;

(* angle of 45 degrees in radians *)
angle := 45.0 * degreesToRadians;
```

Math.sin, Math.cos, Math.tan

```
Math.sin(x : REAL) : REAL     (* sine of x radians        *)
Math.cos(x : REAL) : REAL     (* cosine of x radians      *)
Math.tan(x : REAL) : REAL     (* tangent of x radians     *)
```

The familiar trigonometric ratios:

```
angle := Math.pi / 4.0;    (* 45 degrees *)
Dec.print(Math.sin(angle), 0, 6);  Out.Ln;   (* 0.707107 *)
Dec.print(Math.cos(angle), 0, 6);  Out.Ln;   (* 0.707107 *)
Dec.print(Math.tan(angle), 0, 6);  Out.Ln;   (* 1.000000 *)

Dec.print(Math.sin(Math.pi / 2.0), 0, 6); Out.Ln;  (* 1.000000 -- sin(90°) *)
```

```
Dec.print(Math.cos(Math.pi),       0, 6); Out.Ln;  (* -1.00000 -- cos(180°)*)
```

Note that due to floating-point precision, `Math.cos(Math.pi / 2.0)` will not give exactly 0 — it will give a very small number close to zero, such as `6.12303E-17`. This is normal behaviour for floating-point arithmetic and not a bug.

Math.arcsin, Math.arccos, Math.arctan

```
Math.arcsin(x : REAL) : REAL   (* inverse sine *)
Math.arccos(x : REAL) : REAL   (* inverse cosine *)
Math.arctan(x : REAL) : REAL   (* inverse tangent *)
```

These return angles in radians. For example, to find the angle whose sine is 0.5:

```
angle := Math.arcsin(0.5);
Dec.print(angle * 180.0 / Math.pi, 0, 4);  Out.Ln;  (* 30.0000 -- 30 degrees *)
```

Math.arctan2

```
Math.arctan2(y, x : REAL) : REAL
```

Returns the angle in radians of the point `(x, y)` measured from the positive x-axis, taking into account the signs of both `x` and `y` to determine the correct quadrant. The result is in the range `(-π, π]`. This is more useful than `arctan` for computing angles in two-dimensional geometry, because `arctan` alone cannot distinguish between opposite quadrants:

```
(* Point in the second quadrant: x negative, y positive *)
angle := Math.arctan2(1.0, -1.0);
Dec.print(angle * 180.0 / Math.pi, 0, 4);  Out.Ln;   (* 135.0000 degrees *)

(* Point in the fourth quadrant: x positive, y negative *)
angle := Math.arctan2(-1.0, 1.0);
Dec.print(angle * 180.0 / Math.pi, 0, 4);  Out.Ln;   (* -45.0000 degrees *)
```

18.5 Hyperbolic Functions

The hyperbolic functions are analogues of the trigonometric functions defined using the exponential function rather than the unit circle. They arise in physics, engineering, and the analysis of certain algorithms.

```
Math.sinh(x : REAL) : REAL     (* hyperbolic sine    *)
Math.cosh(x : REAL) : REAL     (* hyperbolic cosine  *)
```

```
Math.tanh(x : REAL) : REAL     (* hyperbolic tangent *)
```

And their inverses:

```
Math.arcsinh(x : REAL) : REAL  (* inverse hyperbolic sine   *)
Math.arccosh(x : REAL) : REAL  (* inverse hyperbolic cosine *)
Math.arctanh(x : REAL) : REAL  (* inverse hyperbolic tangent*)
```

A brief example:

```
Dec.print(Math.sinh(1.0), 0, 6);  Out.Ln;   (* 1.175201 *)
Dec.print(Math.cosh(1.0), 0, 6);  Out.Ln;   (* 1.543081 *)
Dec.print(Math.tanh(1.0), 0, 6);  Out.Ln;   (* 0.761594 *)
```

The hyperbolic functions are less commonly needed in introductory programming, but they are included here for completeness.

18.6 A Reference Table

Here is a complete summary of the `Math` module for quick reference:

Identifier	Type	Description
`Math.pi`	`REAL`	π ≈ 3.14159265358979
`Math.e`	`REAL`	e ≈ 2.71828182845905
`Math.sqrt(x)`	function	Square root of x (x ≥ 0)
`Math.power(b, e)`	function	b raised to the power e
`Math.round(x)`	function	x rounded to nearest integer (as REAL)
`Math.exp(x)`	function	e raised to the power x
`Math.ln(x)`	function	Natural logarithm of x (x > 0)
`Math.log(x, b)`	function	Logarithm of x with base b
`Math.sin(x)`	function	Sine of x (radians)
`Math.cos(x)`	function	Cosine of x (radians)
`Math.tan(x)`	function	Tangent of x (radians)
`Math.arcsin(x)`	function	Inverse sine, result in radians
`Math.arccos(x)`	function	Inverse cosine, result in radians
`Math.arctan(x)`	function	Inverse tangent, result in radians
`Math.arctan2(y,x)`	function	Angle of point (x,y), result in radians
`Math.sinh(x)`	function	Hyperbolic sine
`Math.cosh(x)`	function	Hyperbolic cosine

Identifier	Type	Description
`Math.tanh(x)`	function	Hyperbolic tangent
`Math.arcsinh(x)`	function	Inverse hyperbolic sine
`Math.arccosh(x)`	function	Inverse hyperbolic cosine
`Math.arctanh(x)`	function	Inverse hyperbolic tangent

18.7 Example: Numerical Integration

An **integral** is one of the fundamental concepts of calculus. Informally, the integral of a function `f` over an interval `[a, b]` measures the area between the curve `y = f(x)` and the x-axis, between `x = a` and `x = b`. Areas above the x-axis are positive; areas below are negative.

For simple functions the integral can be computed exactly by hand. For more complex functions — or when writing a general-purpose tool — a numerical approximation is needed. One of the simplest and most effective methods is the **midpoint rule**: divide the interval `[a, b]` into `n` equal strips of width `dx = (b - a) / n`, and approximate the area of each strip as a rectangle whose height is the value of `f` at the midpoint of the strip. Summing the areas of all the rectangles gives an approximation to the integral that improves as `n` increases.

This is an ideal application for procedure types. The function to be integrated varies — sometimes it is `Math.sin`, sometimes `Math.exp`, sometimes a user-defined function — but the integration algorithm itself is always the same. By declaring a procedure type for the function parameter, we can write a single `integral` procedure that works for any function of the right signature.

This example comes directly from Wirth's *Programming in Oberon* tutorial, where it is used to illustrate exactly this point.

```
MODULE Integral;
  IMPORT Out, Math, Real;
TYPE
  Function = PROCEDURE(x : REAL) : REAL;
VAR
  r : REAL;

PROCEDURE integral(f : Function; a, b : REAL; n : INTEGER) : REAL;
VAR
  i      : INTEGER;
  sum, dx : REAL;
BEGIN
  dx  := (b - a) / FLT(n);
  sum := 0.0;
  FOR i := 0 TO n - 1 DO
    sum := f(a + FLT(i) * dx + 0.5 * dx) + sum
  END;
```

```
  RETURN sum * dx
END integral;

BEGIN
  r := integral(Math.sin, 0.0, Math.pi, 20);
  Dec.print(r, 0, 3); Out.Ln;
  r := integral(Math.exp, 0.0, 1.0, 20);
  Dec.print(r, 0, 3); Out.Ln
END Integral.
```

Output:

```
2.002
1.718
```

The procedure type `Function` describes any function procedure that takes a single `REAL` argument and returns a `REAL` result. The `integral` procedure takes such a function as its first parameter `f`, along with the interval endpoints `a` and `b` and the number of strips `n`. Inside the loop, `f(a + FLT(i) * dx + 0.5 * dx)` evaluates whatever function was passed at the midpoint of each strip.

The two calls demonstrate the power of this design. `integral(Math.sin, 0.0, Math.pi, 20)` computes the area under the sine curve from 0 to π — the exact answer is 2, and the approximation with 20 strips gives 2.002. `integral(Math.exp, 0.0, 1.0, 20)` computes the area under `e^x` from 0 to 1 — the exact answer is `e - 1` ≈ `1.718`, which the approximation matches to three decimal places.

Notice that `Math.sin` and `Math.exp` are passed by name, without parentheses or arguments — exactly as with any procedure type variable. They match the `Function` type because they each take one `REAL` argument and return a `REAL` result. Any user-defined function with the same signature can be passed equally:

```
PROCEDURE Square(x : REAL) : REAL;
BEGIN
  RETURN x * x
END Square;

(* integral of x² from 0 to 1 = 1/3 ≈ 0.333 *)
r := integral(Square, 0.0, 1.0, 20);
Dec.print(r, 0, 3); Out.Ln;    (* 0.333 *)
```

The accuracy of the midpoint rule improves with larger `n`. With `n = 100` the sine integral gives 2.000 — indistinguishable from the exact answer at three decimal places.

Summary

- The **`Math`** module provides mathematical constants and functions for use in programs that need more than basic arithmetic.

- The exported constants are `Math.pi` (π) and `Math.e` (e).
- **`Math.sqrt(x)`** returns the square root of `x`. **`Math.power(b, e)`** returns `b` raised to the power `e`. **`Math.round(x)`** returns `x` rounded to the nearest integer as a `REAL`.
- **`Math.exp(x)`** returns e^x. **`Math.ln(x)`** returns the natural logarithm. **`Math.log(x, b)`** returns the logarithm of `x` in base `b`.
- All trigonometric functions work in **radians**. Convert degrees to radians by multiplying by `Math.pi / 180.0`.
- **`Math.sin`**, **`Math.cos`**, **`Math.tan`** are the standard trigonometric functions. Their inverses are **`Math.arcsin`**, **`Math.arccos`**, **`Math.arctan`**, and **`Math.arctan2`** (which handles all four quadrants).
- The hyperbolic functions **`Math.sinh`**, **`Math.cosh`**, **`Math.tanh`** and their inverses are also provided.
- Due to floating-point representation, results may differ very slightly from exact mathematical values — this is normal and expected.
- **Numerical integration** using the midpoint rule is a natural application of procedure types: a single `integral` procedure accepts any function matching a `PROCEDURE(x : REAL) : REAL` signature, including `Math.sin`, `Math.exp`, and user-defined functions.

Exercises

18.1 Write a program that prints a table of sines and cosines for angles from 0° to 360° in steps of 30°. Use `Dec.print` with 4 decimal places. Include a header row showing the angle in degrees, the angle in radians, the sine, and the cosine.

18.2 Write a function procedure `Hypotenuse(a, b : REAL) : REAL` that returns the length of the hypotenuse of a right-angled triangle given the two shorter sides. Use `Math.sqrt` and `Math.power`. Test it with the classic 3-4-5 and 5-12-13 triangles.

18.3 Write a program that reads a positive real number from the user and prints its square root, its natural logarithm, and its base-10 logarithm, all using `Dec.print` with 6 decimal places.

18.4 Write a program that uses `Math.arctan2` to compute and print the angle (in degrees) from the origin to each of the four points (1,1), (-1,1), (-1,-1), and (1,-1). Verify that the results are 45°, 135°, -135°, and -45° respectively.

18.5 The area of a regular polygon with `n` sides and circumradius `r` (the distance from the centre to a corner) is `(n * r² * sin(2π/n)) / 2`. Write a program that reads `n` and `r` from the user and prints the area using `Dec.print` with 4 decimal places. Test it with a hexagon (n=6) and a large value of n (which should approximate the area of a circle, `π * r²`).

18.6 Using the `integral` procedure from Section 18.7, compute and print the following integrals with `n = 50` strips, using `Dec.print` with 6 decimal places. Verify the results against the known exact answers: - `Math.cos` from 0 to π/2 (exact answer: 1) - `Math.sqrt` from 0 to 1 (exact answer: 2/3 ≈ 0.666667) - a function `f(x) = 1.0 / (1.0 + x * x)` from 0 to 1 (exact answer: π/4 ≈ 0.785398)

18.7 Investigate how the accuracy of the midpoint rule changes with `n`. Using `integral(Math.sin, 0.0, Math.pi, n)` (exact answer: 2), print a table showing the result and the absolute error `|result - 2.0|` for `n` = 1, 2, 5, 10, 20, 50, and 100. Use `Dec.print` with 6 decimal places throughout. What do you notice about how the error changes as `n` doubles?

Chapter 19 — The Files Module

Every program we have written so far loses its data when it finishes. Variables exist only in memory — when the program ends, their contents vanish. If you enter a hundred contacts into the address book from Chapter 13 and then close the program, all that work is gone.

Files solve this problem. A file is a named collection of data stored on disk, persisting between program runs. Writing data to a file is called **saving**; reading it back is called **loading**. Together these operations make programs genuinely useful for tasks that involve accumulating or remembering information over time.

OBNC's `Files` module provides a complete set of file operations following the Oakwood Guidelines. This chapter explains how to use them.

19.1 Concepts: Files, Handles, and Riders

Before looking at the procedures, it helps to understand the three key abstractions the `Files` module uses.

Files and Handles

A **file** on disk is identified by its name — a string like `"contacts.dat"` or `"scores.txt"`. From within a program, you do not work with the file directly by name; instead, you work with a **handle** — an object of type `Files.File` that represents the open file. Think of a handle as a remote control for the file: once you have one, you can read from it, write to it, or query its properties, without using the filename again.

`Files.File` is declared as `POINTER TO Handle` in the module. You declare a variable of this type and obtain a handle by calling `Files.Old` (to open an existing file) or `Files.New` (to create a new one). A handle value of `NIL` means no file is open — either the file was not found or something went wrong.

Riders

A **rider** is a read/write position marker attached to a file. It is a record variable of type `Files.Rider` that keeps track of where in the file the next read or write will take place. You can think of it as a cursor — like the blinking cursor in a text editor, but for a file.

A rider is positioned with `Files.Set(r, f, pos)`, which attaches rider `r` to file `f` at byte position `pos`. Position `0` is the start of the file. After each read or write operation, the rider automatically advances to the next position.

The rider record has two important fields: - `r.eof` — set to `TRUE` when a read operation reaches the end of the file. - `r.res` — set to `0` when an operation succeeds; non-zero on partial or failed operations.

You declare a rider as a plain `VAR` — it is a record, not a pointer, so no `NEW` is needed:

```
VAR
  f : Files.File;
  r : Files.Rider;
```

19.2 Opening and Creating Files

Files.Old — Opening an Existing File

```
Files.Old(name : ARRAY OF CHAR) : Files.File
```

Searches for a file with the given name and returns a handle to it. If the file does not exist, returns `NIL`. Always check for `NIL` before using the result:

```
f := Files.Old("data.txt");
IF f = NIL THEN
  Out.String("File not found.");
  Out.Ln
ELSE
  (* proceed to read *)
END
```

Files.New — Creating a New File

```
Files.New(name : ARRAY OF CHAR) : Files.File
```

Creates a new file in memory and returns a handle to it. The name is remembered for later use but the file is **not yet written to disk** — it exists only in memory until `Files.Register` is called. This two-step create-then-register approach means a partially written file is never left on disk if the program crashes during writing.

```
f := Files.New("output.dat");
```

Files.Register — Writing a New File to Disk

```
Files.Register(f : Files.File)
```

Writes the file `f` to disk under the name provided to `Files.New`. If a file with that name already exists, it is overwritten. This is the step that makes the file permanent:

```
f := Files.New("output.dat");
(* ... write data to f using a rider ... *)
Files.Register(f);    (* now the file exists on disk *)
```

Files.Close — Flushing Buffers

```
Files.Close(f : Files.File)
```

Writes any buffered data back to disk without removing the file from the directory. For files opened with `Files.Old` and modified, `Files.Close` ensures the changes are saved. For files created with `Files.New`, `Files.Register` should be called instead (which also flushes the buffers).

19.3 Setting Up a Rider

Before reading from or writing to a file, you must set a rider to the desired position:

```
Files.Set(r, f, pos)
```

`r` is the rider (passed by `VAR`), `f` is the file handle, and `pos` is the byte position (`0` = start of file). To read or write from the beginning:

```
Files.Set(r, f, 0);
```

To append to the end of an existing file:

```
Files.Set(r, f, Files.Length(f));
```

`Files.Length(f)` returns the total number of bytes in the file.

19.4 Writing to a File

Once a rider is positioned, writing is straightforward. Each `Write` procedure takes the rider as a `VAR` parameter and advances it after writing.

Procedure	What it writes
`Files.Write(r, x)`	One byte (`x : BYTE`)
`Files.WriteInt(r, i)`	One integer (`i : INTEGER`)
`Files.WriteReal(r, x)`	One real number (`x : REAL`)
`Files.WriteString(r, s)`	A string including its `0X` terminator
`Files.WriteBool(r, b)`	A Boolean value (`b : BOOLEAN`)
`Files.WriteSet(r, s)`	A set value (`s : SET`)
`Files.WriteNum(r, i)`	An integer, compactly encoded
`Files.WriteBytes(r, buf, n)`	`n` bytes from array `buf`

Example 19.1: Writing Integers to a File

```
MODULE WriteInts;
  IMPORT Out, Files;
VAR
  f : Files.File;
  r : Files.Rider;
  i : INTEGER;
BEGIN
  f := Files.New("numbers.dat");
  Files.Set(r, f, 0);
  FOR i := 1 TO 5 DO
    Files.WriteInt(r, i * 10)
  END;
  Files.Register(f);
  Out.String("Written successfully.");
  Out.Ln
END WriteInts.
```

This creates `numbers.dat` containing the five integers 10, 20, 30, 40, 50, stored in binary form.

19.5 Reading from a File

Reading mirrors writing. Each `Read` procedure takes the rider as `VAR` and a `VAR` parameter to receive the value read:

Procedure	What it reads
`Files.Read(r, x)`	One byte into `x : BYTE`
`Files.ReadInt(r, i)`	One integer into `i : INTEGER`
`Files.ReadReal(r, x)`	One real into `x : REAL`
`Files.ReadString(r, s)`	A null-terminated string into `s`
`Files.ReadBool(r, b)`	A Boolean into `b : BOOLEAN`
`Files.ReadSet(r, s)`	A set into `s : SET`
`Files.ReadNum(r, i)`	A compactly encoded integer
`Files.ReadBytes(r, buf, n)`	`n` bytes into array `buf`

After each read, check `r.eof` to detect the end of the file before attempting the next read.

Example 19.2: Reading Integers Back

```
MODULE ReadInts;
  IMPORT Out, Files;
```

```
VAR
  f : Files.File;
  r : Files.Rider;
  i : INTEGER;
BEGIN
  f := Files.Old("numbers.dat");
  IF f = NIL THEN
    Out.String("File not found.");
    Out.Ln
  ELSE
    Files.Set(r, f, 0);
    Files.ReadInt(r, i);
    WHILE ~r.eof DO
      Out.Int(i, 0);
      Out.Ln;
      Files.ReadInt(r, i)
    END
  END
END ReadInts.
```

Output:

```
10
20
30
40
50
```

The pattern `ReadInt; WHILE ~r.eof DO ... ReadInt END` is the standard idiom for reading until the end of a file. The read is called once before the loop to prime the first value, and again at the end of the loop body to advance to the next. This mirrors the `In.Int; WHILE In.Done DO ... In.Int END` pattern from earlier chapters.

19.6 The Binary Format

An important point: `Files.WriteInt` and `Files.ReadInt` store integers in **binary** format — the raw bytes of the integer's internal representation, not its decimal digits. A file written with `WriteInt` cannot be opened and read meaningfully in a text editor. This is normal for data files, but it means the write and read procedures must match exactly: data written with `WriteInt` must be read back with `ReadInt`, data written with `WriteReal` must be read back with `ReadReal`, and so on.

If you need human-readable files that can be opened in a text editor, write strings using `WriteString` and the `extConvert` module from Chapter 17 to convert numbers to strings first.

19.7 Other File Operations

Files.Delete

```
Files.Delete(name : ARRAY OF CHAR; VAR res : INTEGER)
```

Removes the named file from the directory. `res` is 0 on success, non-zero on failure (for example, if the file does not exist):

```
Files.Delete("temp.dat", res);
IF res # 0 THEN
  Out.String("Could not delete file.");
  Out.Ln
END
```

Files.Rename

```
Files.Rename(old, new : ARRAY OF CHAR; VAR res : INTEGER)
```

Renames the file `old` to `new`. `res` is 0 on success:

```
Files.Rename("draft.dat", "final.dat", res);
```

Files.Length

```
Files.Length(f : Files.File) : INTEGER
```

Returns the total number of bytes in file `f`. Useful for computing the number of records stored: for example, if each record is one integer (8 bytes on a 64-bit system), `Files.Length(f) DIV 8` gives the record count.

Files.Pos

```
Files.Pos(VAR r : Files.Rider) : INTEGER
```

Returns the current byte position of rider `r`.

Files.GetError

```
Files.GetError(VAR msg : ARRAY OF CHAR)
```

Returns a descriptive error message for the last failed file operation. Useful for diagnosing problems when `r.res` is non-zero or a handle is `NIL`.

19.8 Saving and Loading a List of Names

We will write two programs: one that reads names from the user and saves them to a file, and one that loads and prints those names. Together they demonstrate the complete save/load cycle.

The key design decision is the file format. We will write each name as a string using `Files.WriteString`, which stores the characters including the null terminator. `Files.ReadString` reads them back in the same way. This is simple and reliable, though as noted above the file is in binary format. The file is not a plain text file — each name is separated by a null terminator byte `0X` rather than a newline, so it would appear garbled in a text editor. If you need a file that opens cleanly as plain text, you would use `Files.WriteString` with manually appended newline characters, or redirect output to a file at the command line (see section 19.10 below).

Saving Names

```
MODULE SaveNames;
  IMPORT In, Out, Files;
CONST
  NameLen  = 30;
  FileName = "names.dat";
TYPE
  NameStr = ARRAY NameLen OF CHAR;
VAR
  f     : Files.File;
  r     : Files.Rider;
  name  : NameStr;
  flush : ARRAY 10 OF CHAR;
  count : INTEGER;
BEGIN
  f := Files.New(FileName);
  Files.Set(r, f, 0);
  count := 0;
  Out.String("Enter names, one per line.");
  Out.Ln;
  Out.String("Press Enter on a blank line to finish.");
  Out.Ln;
  REPEAT
    Out.String("Name: ");
    In.Line(name);
    IF name[0] # 0X THEN
      Files.WriteString(r, name);
      INC(count)
    END
  UNTIL name[0] = 0X;
```

```
  Files.Register(f);
  Out.String("Saved ");
  Out.Int(count, 0);
  Out.String(" name(s) to ");
  Out.String(FileName);
  Out.Ln
END SaveNames.
```

A sample run:

```
Enter names, one per line.
Press Enter on a blank line to finish.
Name: Alice
Name: Bob
Name: Carol
Name:
Saved 3 name(s) to names.dat
```

Loading Names

```
MODULE LoadNames;
  IMPORT Out, Files;
CONST
  NameLen  = 30;
  FileName = "names.dat";
TYPE
  NameStr = ARRAY NameLen OF CHAR;
VAR
  f     : Files.File;
  r     : Files.Rider;
  name  : NameStr;
  count : INTEGER;
BEGIN
  f := Files.Old(FileName);
  IF f = NIL THEN
    Out.String("File '");
    Out.String(FileName);
    Out.String("' not found.");
    Out.Ln
  ELSE
    Files.Set(r, f, 0);
    count := 0;
    Files.ReadString(r, name);
    WHILE ~r.eof DO
```

```
      Out.String(name);
      Out.Ln;
      INC(count);
      Files.ReadString(r, name)
    END;
    Out.Ln;
    Out.String("Loaded ");
    Out.Int(count, 0);
    Out.String(" name(s).");
    Out.Ln
  END
END LoadNames.
```

Output (after running `SaveNames` first):

```
Alice
Bob
Carol

Loaded 3 name(s).
```

How It Works

`SaveNames` creates a new file, sets a rider at position 0, and writes each name as a string. `Files.WriteString` writes the characters of the name plus the null terminator `0X` — that terminator is what allows `ReadString` to know where one name ends and the next begins.

`LoadNames` opens the same file with `Files.Old`, sets a rider at position 0, and reads strings in a loop until `r.eof` becomes `TRUE`. The read-before-loop pattern ensures that `r.eof` is set before it is tested.

Note that the two programs must agree on the file format — both use `WriteString`/`ReadString` with the same `NameLen`. If you change one, you must change the other, and any existing data files become unreadable until regenerated. This is the fundamental contract of binary file formats.

19.9 Saving and Loading Records

Saving a complete record to a file requires writing each field individually. There is no single `WriteRecord` procedure — you write field by field, and read field by field in the same order:

```
(* Writing a Contact record *)
Files.WriteString(r, contact.name);
Files.WriteString(r, contact.phone);
Files.WriteInt(r, contact.age);

(* Reading it back *)
Files.ReadString(r, contact.name);
```

```
Files.ReadString(r, contact.phone);
Files.ReadInt(r, contact.age);
```

The order must be identical in both directions. A natural way to ensure this is to write `SaveContact` and `LoadContact` procedures side by side in the source code, making their symmetry visually obvious:

```
PROCEDURE SaveContact(VAR r : Files.Rider; VAR c : Contact);
BEGIN
  Files.WriteString(r, c.name);
  Files.WriteString(r, c.phone);
  Files.WriteInt(r, c.age)
END SaveContact;

PROCEDURE LoadContact(VAR r : Files.Rider; VAR c : Contact);
BEGIN
  Files.ReadString(r, c.name);
  Files.ReadString(r, c.phone);
  Files.ReadInt(r, c.age)
END LoadContact;
```

This paired-procedure pattern is the standard approach for saving and loading structured data with the `Files` module, and we will use it in the address book project in Chapter 23.

19.10 Input and Output Redirection

The `Files` module is the right tool when a program needs fine-grained control over file operations — creating, renaming, or appending to specific files, saving structured records, or managing multiple files at once. But sometimes you simply want to read input from a file instead of the keyboard, or save output to a file instead of printing it to the screen. For these cases, the operating system provides a simpler mechanism: **I/O redirection**.

When you run an OBNC program from the terminal, the standard input (what `In` reads from) and standard output (what `Out` writes to) are both connected to the terminal by default. The shell lets you redirect either or both to files using the < and > operators, without changing a single line of your program.

Redirecting Output

To save everything a program prints to a file, use > followed by the filename:

```
./MyProgram > output.txt
```

The program runs normally, but instead of appearing on the screen, all output from Out is written to `output.txt`. If the file already exists it is overwritten; if it does not exist it is created. To **append** to an existing file instead of overwriting it, use >>:

```
./MyProgram >> output.txt
```

Redirecting Input

To make a program read from a file instead of the keyboard, use <:

```
./MyProgram < input.txt
```

The program calls `In.Int`, `In.Line`, and so on as usual, but they read from `input.txt` rather than waiting for keyboard input. The file must contain data in exactly the format the program expects — the same values you would have typed, one per line.

Combining Both

You can redirect input and output simultaneously:

```
./MyProgram < input.txt > output.txt
```

The program reads from `input.txt` and writes to `output.txt`, with no interaction with the terminal at all. This is useful for batch processing — running the same program repeatedly with different input files — and for testing, since you can prepare a known input file and check the output file against expected results.

A Practical Example

Recall the `Stats` program from Chapter 8, which reads a list of numbers and prints statistics. Instead of typing the numbers each time you run it, you could prepare a file `data.txt` containing:

```
3.0
7.5
2.0
9.5
4.0
```

and run:

```
./Stats < data.txt
```

The program reads the numbers from the file exactly as if they had been typed. To save the results as well:

```
./Stats < data.txt > results.txt
```

When to Use Redirection vs the Files Module

Redirection is ideal when the program already uses `In` and `Out` and you just want to point them at files rather than the terminal. It requires no changes to the program at all.

The `Files` module is needed when you want to read and write specific named files from within the program itself — for example, automatically loading a contacts file at startup without the user having to redirect input manually, writing to multiple files, appending to an existing file under program control, or saving structured binary data. For anything more than simple sequential reading and writing, `Files` is the right choice.

Summary

- A **file** provides persistent storage that survives between program runs. In OBNC, files are accessed through the `Files` module.
- A **handle** (`Files.File`) represents an open file. Obtain one with `Files.Old` (existing file) or `Files.New` (new file). A `NIL` handle means the file could not be opened or created.
- A **rider** (`Files.Rider`) is a position marker attached to a file with `Files.Set(r, f, pos)`. It advances automatically after each read or write. Check `r.eof` to detect the end of the file during reading.
- New files created with `Files.New` are not written to disk until `Files.Register` is called. This protects against partially written files.
- Write and read procedures come in matched pairs: `WriteInt`/`ReadInt`, `WriteReal`/`ReadReal`, `WriteString`/`ReadString`, and so on. The format used for writing must exactly match the format used for reading.
- The standard read-loop idiom is: read once before the loop, then `WHILE ~r.eof DO ... read again END`.
- Records are saved and loaded field by field. Use paired `Save` and `Load` procedures to keep the field order consistent.
- `Files.Delete` and `Files.Rename` manage files in the directory. `Files.Length` returns the file size in bytes. `Files.GetError` returns a descriptive message for the last failed operation.

Exercises

19.1 Write a program `WriteReals` that writes 5 real numbers entered by the user to a file `reals.dat` using `Files.WriteReal`. Write a corresponding program `ReadReals` that reads and prints them back using `Files.ReadReal` and `Dec.print` with 4 decimal places.

19.2 Write a program that saves an array of 10 integers (the squares of 1 to 10) to a file, then reads them back and prints them. Use a single program with two procedures: `SaveArray` and `LoadArray`.

19.3 Extend the `SaveNames`/`LoadNames` example to also save and load the count of names as the first item in the file using `Files.WriteInt` and `Files.ReadInt`. This allows `LoadNames` to allocate exactly the right number of reads rather than looping until `r.eof`.

19.4 Write a program that reads a text file line by line using `In` (redirect standard input from the file with `./program < input.txt`) and counts the number of lines, words, and characters. Print the three counts.

19.5 Write a program `AppendName` that opens an existing `names.dat` file (created by `SaveNames`) and appends one new name to it. Use `Files.Set(r, f, Files.Length(f))` to position the rider at the end before writing. Verify by running `LoadNames` afterwards.

19.6 Extend the `Contacts` program from Chapter 13 to save the contact list to a file when the user finishes entering data, and load it at startup if the file already exists. Use the paired `SaveContact`/`LoadContact` procedure pattern from Section 19.9. The program should feel seamless — contacts entered in a previous run should still be present when the program is run again.

19.7 Write a program that reads an existing `names.dat` file, sorts the names alphabetically using selection sort, and writes the sorted names back to the same file. Print the sorted list to the screen as well. (Hint: load all names into an array first, sort the array, then rewrite the file from scratch using `Files.New` and `Files.Register`.)

Chapter 20 — Error Handling & Assertions

Every program makes assumptions. A procedure that computes a square root assumes its argument is non-negative. A binary search assumes its array is sorted. A file loader assumes the file exists. These assumptions are usually met — but not always. When they are violated, the program may produce wrong results, crash in a confusing way, or silently corrupt data.

Defensive programming is the practice of making assumptions explicit and handling violations gracefully. This chapter introduces Oberon-07's tools for doing so: the `ASSERT` statement, the runtime trap system, the `extTrap` module for customising trap behaviour, and the `extErr` module for writing diagnostic messages to the standard error stream.

20.1 The ASSERT Statement

The simplest and most important defensive programming tool in Oberon-07 is `ASSERT`. It checks that a condition is true, and if it is not, stops the program immediately with a trap.

The syntax is:

```
ASSERT(condition)
ASSERT(condition, errorCode)
```

The first form uses a default error code of 0. The second form lets you specify your own integer error code, which appears in the trap message and can be used by a custom trap handler (see Section 20.3) to identify what went wrong.

```
ASSERT(x >= 0.0);              (* trap if x is negative        *)
ASSERT(n > 0, 100);            (* trap with code 100 if n <= 0 *)
ASSERT(p # NIL);               (* trap if pointer is NIL       *)
```

`ASSERT` is not just for debugging. It is the idiomatic Oberon-07 way to express a **precondition** — a condition that must be true for a procedure to work correctly. Placed at the top of a procedure, assertions document what the procedure requires and enforce those requirements at runtime:

```
PROCEDURE Sqrt(x : REAL) : REAL;
BEGIN
  ASSERT(x >= 0.0);
  RETURN Math.sqrt(x)
END Sqrt;
```

If `Sqrt` is ever called with a negative argument, the `ASSERT` fires immediately, the trap message identifies the location in the source file, and the programmer knows exactly where the problem is. Without the assertion, `Math.sqrt` would receive a negative argument and produce a `NaN` (not-a-number) result that might propagate silently through the program, making the bug very hard to find.

ASSERT vs IF

A common question is: when should you use `ASSERT` rather than an `IF` statement?

Use `ASSERT` when a condition represents a **programming error** — something that should never happen if the program is correct. The assumption that `x >= 0.0` in `Sqrt` is the caller's responsibility; if it is violated, the program has a bug. `ASSERT` makes that bug visible immediately.

Use `IF` when a condition represents a **runtime situation** the program should handle gracefully — for example, a file not found, invalid user input, or a value out of an expected range. These are not bugs; they are expected possibilities that the program should deal with sensibly.

```
(* Bug prevention -- use ASSERT *)
PROCEDURE SetAge(VAR p : Person; age : INTEGER);
BEGIN
  ASSERT(age >= 0);
  ASSERT(age <= 150);
  p.age := age
END SetAge;

(* Expected condition -- use IF *)
f := Files.Old("data.txt");
IF f = NIL THEN
  Out.String("File not found. Starting with empty data.");
  Out.Ln
ELSE
  LoadData(f)
END
```

ASSERT(FALSE)

A special use of `ASSERT` is `ASSERT(FALSE)`, which always traps. This is useful for marking code paths that should never be reached:

```
CASE choice OF
  1 : DoOption1 |
  2 : DoOption2 |
  3 : DoOption3
ELSE
  ASSERT(FALSE)    (* should never reach here *)
END
```

If the `CASE` statement somehow receives an unexpected value, the `ASSERT(FALSE)` fires immediately rather than silently doing nothing.

20.2 Runtime Traps and Exception Codes

When something goes wrong at runtime — an array index out of bounds, a `NIL` pointer dereference, a failed assertion — Oberon-07 raises a **trap**. By default, the program stops and prints a message such as:

```
prog: src/MyModule.obn:42: assertion failure (8)
```

This tells you the program name, the source file, the line number, and the exception code. The exception codes defined by OBNC are:

Code	Meaning
1	Destination array too short for assignment
2	Array index out of bounds
3	NIL pointer dereference
4	NIL procedure variable call
5	Source in assignment is not an extension of target
6	Type guard failure
7	Unmatched expression in CASE statement
8	Assertion failure

These codes are your first clue when a program crashes. Code 2 (index out of bounds) usually means a loop ran one iteration too many. Code 3 (NIL dereference) means you followed a pointer without checking for `NIL` first. Code 8 (assertion failure) means a precondition was violated.

The default trap message is informative enough for development, but in a finished program you may want a more user-friendly response. That is where `extTrap` comes in.

20.3 The extTrap Module

The `extTrap` module allows you to install a custom **trap handler** — a procedure that is called instead of the default message when a trap occurs. This lets you control exactly what happens when something goes wrong: print a friendly message, log the error to a file, attempt a recovery, or terminate cleanly.

```
IMPORT extTrap;
```

The module exports one type, one variable, and one procedure:

```
TYPE
  Handler = PROCEDURE(exception : INTEGER; file : ARRAY OF CHAR; line : INTEGER);

VAR
```

```
  handle : Handler;    (* the current trap handler -- read-only in clients *)

PROCEDURE SetHandler(h : Handler);
```

`Handler` is a procedure type — exactly as discussed in Chapter 14. The handler procedure receives the exception code, the source file name, and the line number where the trap occurred. You write a procedure matching this signature and register it with `extTrap.SetHandler`.

Example 20.1: A Custom Trap Handler

```
MODULE TrapDemo;
  IMPORT Out, extErr, extTrap;
VAR
  a : ARRAY 5 OF INTEGER;
  i : INTEGER;

PROCEDURE MyHandler(exception : INTEGER;
                    file      : ARRAY OF CHAR;
                    line      : INTEGER);
BEGIN
  extErr.String("Error in ");
  extErr.String(file);
  extErr.String(" at line ");
  extErr.Int(line, 0);
  extErr.String(": ");
  IF exception = 2 THEN
    extErr.String("array index out of bounds")
  ELSIF exception = 3 THEN
    extErr.String("NIL pointer dereference")
  ELSIF exception = 8 THEN
    extErr.String("assertion failure")
  ELSE
    extErr.String("exception code ");
    extErr.Int(exception, 0)
  END;
  extErr.Ln;
  ASSERT(FALSE)    (* re-raise to terminate the program *)
END MyHandler;

BEGIN
  extTrap.SetHandler(MyHandler);
  FOR i := 0 TO 5 DO    (* bug: should be 0 TO 4 *)
    a[i] := i * 10
  END;
  Out.String("Done.");
```

```
  Out.Ln
END TrapDemo.
```

Output to standard error:

```
Error in TrapDemo.obn at line 30: array index out of bounds
```

The `ASSERT(FALSE)` at the end of the handler re-raises a trap to terminate the program after the message has been printed. Without it, the program would attempt to continue after the trap, which is usually not safe.

Note that the error message is written to standard error (using `extErr`) rather than standard output (using `Out`). This is the correct practice for error messages, as discussed in the next section.

When to Use a Custom Handler

For simple programs, the default trap message is perfectly adequate. A custom handler is most useful when:

- You want more descriptive messages than the default provides.
- You need to clean up resources (close files, save state) before the program terminates.
- You are writing a library module and want to provide context-specific error messages to clients.

20.4 The extErr Module

Standard output (`Out`) and standard error (`extErr`) are two separate output streams. By convention:

- **Standard output** (`Out`) carries the program's normal results — the data the user asked for.
- **Standard error** (`extErr`) carries diagnostic messages — warnings, error reports, and debugging information.

The reason for this separation is practical. When a user redirects a program's output to a file (`./myprogram > results.txt`), standard output goes to the file but standard error still appears on the terminal. This means error messages are always visible to the user, even when normal output is being captured.

```
IMPORT extErr;
```

The `extErr` module mirrors `Out` almost exactly:

Procedure	Description
`extErr.Char(ch)`	Write a character to standard error
`extErr.String(s)`	Write a string to standard error
`extErr.Int(i, n)`	Write an integer, right-justified in width `n`
`extErr.Real(x, n)`	Write a real in scientific notation

Procedure	Description
`extErr.Hex(i)`	Write an integer as zero-padded hex
`extErr.Ln`	Write a newline to standard error

Usage is identical to `Out`:

```
extErr.String("Warning: value out of expected range: ");
extErr.Int(value, 0);
extErr.Ln;
```

A Simple Convention

A consistent convention for error reporting in OBNC programs is:

- Use `Out` for all normal program output.
- Use `extErr` for all warnings, errors, and diagnostic messages.
- In trap handlers, always use `extErr`, never `Out`.

This keeps the two streams cleanly separated and makes programs behave correctly under output redirection.

20.5 Example: A Robust Input Validator

Let us bring everything together in a program that validates user input defensively. The program reads examination scores (which must be in the range 0 to 100) from the user, computes the mean, and reports the result. We want it to:

- Stop when the user enters a non-number, detected via `In.Done` becoming false.
- Reject scores outside [0, 100] with a clear message on standard error, and continue asking for the next score without recording the invalid entry.
- Use `ASSERT` internally to verify its own logic.
- Write all error messages to standard error.

```
MODULE ScoreStats;
  IMPORT In, Out, extErr, extTrap;
CONST
  MaxScores = 20;
  MinScore  = 0;
  MaxScore  = 100;
VAR
  scores : ARRAY MaxScores OF INTEGER;
  n      : INTEGER;
  score  : INTEGER;
  done   : BOOLEAN;

PROCEDURE MyHandler(exception : INTEGER;
```

```
                    file      : ARRAY OF CHAR;
                    line      : INTEGER);
BEGIN
  extErr.String("Internal error (");
  extErr.Int(exception, 0);
  extErr.String(") in ");
  extErr.String(file);
  extErr.String(" line ");
  extErr.Int(line, 0);
  extErr.Ln;
  ASSERT(FALSE)
END MyHandler;

PROCEDURE ComputeMean(VAR s : ARRAY OF INTEGER; count : INTEGER) : REAL;
VAR
  i, total : INTEGER;
BEGIN
  ASSERT(count > 0);
  total := 0;
  FOR i := 0 TO count - 1 DO
    INC(total, s[i])
  END;
  RETURN FLT(total) / FLT(count)
END ComputeMean;

BEGIN
  extTrap.SetHandler(MyHandler);
  n    := 0;
  done := FALSE;
  Out.String("Enter scores (0-100). Enter a non-number to finish.");
  Out.Ln;
  WHILE (n < MaxScores) & ~done DO
    Out.String("Score ");
    Out.Int(n + 1, 0);
    Out.String(": ");
    In.Int(score);
    IF ~In.Done THEN
      done := TRUE
    ELSIF (score < MinScore) OR (score > MaxScore) THEN
      extErr.String("Error: score must be between ");
      extErr.Int(MinScore, 0);
      extErr.String(" and ");
      extErr.Int(MaxScore, 0);
      extErr.String(". Not recorded.");
      extErr.Ln
    ELSE
```

```
      scores[n] := score;
      INC(n)
    END
  END;
  IF n = 0 THEN
    extErr.String("No valid scores entered.");
    extErr.Ln
  ELSE
    Out.Ln;
    Out.String("Scores entered: ");
    Out.Int(n, 0);
    Out.Ln;
    Out.String("Mean score:     ");
    Out.Real(ComputeMean(scores, n), 0);
    Out.Ln
  END
END ScoreStats.
```

A sample run:

```
Enter scores (0-100). Enter a non-number to finish.
Score 1: 85
Score 2: 92
Score 3: 150
Error: score must be between 0 and 100. Not recorded.
Score 3: 78
Score 4: done

Scores entered: 3
Mean score:     8.50000E+01
```

Out-of-range scores produce a clear error message on standard error and are simply ignored — the loop continues asking for the next score without advancing `n`. Valid scores are stored and counted. When the user enters a non-number (`done` in the sample run), `In.Done` becomes false, the `done` flag is set, and the loop exits cleanly.

The `ASSERT(count > 0)` precondition in `ComputeMean` is never triggered in normal use — the `IF n = 0` check in the main block ensures `ComputeMean` is only called when at least one score has been recorded. However, if a bug elsewhere caused `ComputeMean` to be called with `n = 0`, the custom handler would produce a clear diagnostic rather than a division by zero or a garbage result. This is exactly the role of `ASSERT`: not to handle expected situations, but to catch programming errors that should never occur.

Summary

- **`ASSERT(condition)`** checks that a condition holds at runtime. If it does not, the program traps

with exception code 8. Use `ASSERT` to express preconditions — conditions that must be true for the program to be correct. Use `IF` for expected runtime situations the program should handle gracefully.

- **`ASSERT(condition, code)`** traps with a specific integer error code, useful for identifying which assertion failed in a custom trap handler.
- **`ASSERT(FALSE)`** always traps. Use it to mark code paths that should never be reached, and to terminate a custom trap handler after printing a message.
- The **OBNC exception codes** are: 1 (array too short), 2 (index out of bounds), 3 (NIL dereference), 4 (NIL procedure call), 5 (bad assignment), 6 (type guard failure), 7 (unmatched CASE), 8 (assertion failure).
- The **`extTrap`** module allows you to install a custom trap handler using `extTrap.SetHandler(h)`. The handler is a procedure matching the `Handler` type: `PROCEDURE(exception : INTEGER; file : ARRAY OF CHAR; line : INTEGER)`. End the handler with `ASSERT(FALSE)` to terminate the program after printing the message.
- The **`extErr`** module writes to standard error — a separate output stream from `Out`. Error messages written with `extErr` remain visible on the terminal even when standard output is redirected to a file.
- Use `Out` for normal program output; use `extErr` for all error and diagnostic messages.

Exercises

20.1 Add `ASSERT` statements to the `BinarySearch` procedure from Chapter 15 to check that `low >= 0` and `high < LEN(a)` at the start of each call. Run the program with valid and invalid inputs to verify the assertions fire correctly.

20.2 Write a procedure `SafeDivide(a, b : INTEGER; VAR result : INTEGER; VAR ok : BOOLEAN)` that divides `a` by `b`, sets `ok := TRUE` and stores the result if `b # 0`, and sets `ok := FALSE` with an `extErr` message if `b = 0`. Write a driver to test it.

20.3 Write a function procedure `GetInRange(lo, hi : INTEGER) : INTEGER` that repeatedly prompts the user until they enter an integer in the range `[lo, hi]`, printing an `extErr` message for each invalid entry. Use `ASSERT(lo <= hi)` as a precondition. Test it with several ranges.

20.4 Install a custom trap handler that prints a descriptive message for each of the eight exception codes. Test it by deliberately triggering at least three different exceptions — an out-of-bounds array access, a `NIL` dereference, and a failed assertion — in separate programs.

20.5 Extend the `ScoreStats` program to also print the minimum and maximum scores alongside the mean. Add `ASSERT` statements to your `Min` and `Max` procedures to verify that `count > 0` before they attempt to scan the array.

20.6 Write a program that reads a filename from the command line using `extArgs`, opens the file, and reads integers from it one by one using the `Files` module. Use `extErr` to print a friendly message if the file cannot be opened, followed by `ASSERT(FALSE)` to terminate. Test with both a valid and a missing file.

Chapter 21 — Miscellaneous Topics

This chapter covers three topics that do not fit neatly into the earlier structure of the book but are useful in practice: interfacing Oberon-07 programs to C libraries, inter-process communication using the `extPipes` module, and graphics programming with the `XYplane`, `RectUI`, and `XYtext` modules.

21.1 Interfacing to C

Oberon-07 is a clean, safe language, but it's fairly bare-bones and there are times when you need to call functionality that is not available in it — operating system calls, hardware interfaces, or existing C libraries. OBNC makes this possible through a straightforward C interfacing mechanism.

How It Works

When OBNC compiles a module `Foo.obn`, it looks for a companion file `Foo.c` in the same directory. If found, the C file is compiled and linked together with the Oberon module. Procedures declared in the Oberon module without a body — ending with the declaration line rather than a `BEGIN`…`END` block — are expected to be implemented in the C file.

The naming convention is important: an Oberon procedure `Foo.Bar` corresponds to a C function named `Foo__Bar_` (two underscores before the procedure name, one trailing underscore). OBNC handles the calling convention and type mapping automatically for basic types.

The type mappings between Oberon and C are:

Oberon type	C type
`INTEGER`	`OBNC_INTEGER`
`REAL`	`OBNC_REAL`
`BOOLEAN`	`int`
`CHAR`	`char`
`BYTE`	`unsigned char`
`SET`	`OBNC_INTEGER`

The OBNC header `<obnc/OBNC.h>` defines `OBNC_INTEGER` and `OBNC_REAL` to match the sizes used by the Oberon compiler on your platform.

Every C companion file must also include the generated header for the module (`.obnc/ModuleName.h`) and define an initialisation function `ModuleName__Init(void)`, even if it does nothing. This function is called automatically when the module is loaded.

Example 21.1: The Delay Module

As a practical example we will create a `Delay` module that provides two utilities implemented in C: a millisecond-precision sleep function using the standard `usleep` function, and a flush function that forces any buffered output to appear on the terminal immediately.

The need for flushing arises because standard output is normally **line-buffered** — output is only

sent to the terminal when a newline character is written. If you print several values on the same line with pauses between them, the values may all appear at once at the end rather than one by one. Calling `Delay.Flush` after each value forces the buffer to be emptied immediately.

The Oberon module — `Delay.obn`:

```
MODULE Delay;

PROCEDURE Sleep*(ms : INTEGER);
  (* waits for approximately ms milliseconds *)
END Sleep;

PROCEDURE Flush*;
END Flush;

END Delay.
```

Both procedures are declared with export markers `*` and have no bodies — no `BEGIN...END` block — which tells OBNC to look for their implementations in the companion C file.

The C implementation — `Delay.c`:

```
#include ".obnc/Delay.h"
#include <obnc/OBNC.h>
#include <unistd.h>

#define OBERON_SOURCE_FILENAME "Delay.obn"

void Delay__Sleep_(OBNC_INTEGER ms_)
{
    usleep((useconds_t)ms_ * 1000);
}

void Delay__Flush_(void)
{
    fflush(stdout);
}

void Delay__Init(void)
{
}
```

`Delay__Sleep_` calls `usleep` with the millisecond value converted to microseconds (multiply by 1000). `Delay__Flush_` calls `fflush(stdout)` to flush the standard output buffer. The `Delay__Init` function is required but has nothing to do.

Both files must be in the same directory. When any program importing `Delay` is compiled with `obnc`,

the C file is detected and compiled automatically — no separate compilation step is needed.

Using the Delay Module

The following countdown program uses `Delay.Sleep` to pause one second between each number. Because all the numbers appear on the same line, `Delay.Flush` is called after each number to ensure it appears on the terminal immediately rather than being held in the output buffer:

```
MODULE countDown;
  IMPORT Out, Delay;
VAR
  i : INTEGER;
BEGIN
  FOR i := 10 TO 1 BY -1 DO
    Out.Int(i, 3);
    Delay.Flush;
    Delay.Sleep(1000)
  END;
  Out.String("  Blast off!");
  Out.Ln
END countDown.
```

Output (with one second between each number):

```
 10  9  8  7  6  5  4  3  2  1  Blast off!
```

If each number were printed on its own line with `Out.Ln`, the `Delay.Flush` call would not be necessary — the newline itself flushes the buffer. The flush is only needed when output on the same line must appear incrementally.

Summary of Steps

Interfacing a C function to Oberon requires the following steps:

1. **Write the Oberon module** (`ModuleName.obn`). Declare each procedure that will be implemented in C without a body — just the procedure header with a comment describing what it does, followed immediately by `END ProcedureName;`. Export any procedures that client modules will call by marking them with `*`.
2. **Write the C implementation** (`ModuleName.c`). Include the generated module header `.obnc/ModuleName.h` and the `OBNC` runtime header `<obnc/OBNC.h>`, plus any standard C headers you need. Implement each bodyless Oberon procedure as a C function named `ModuleName__ProcedureName_` (two underscores before the procedure name, one trailing underscore). Use `OBNC_INTEGER` for `INTEGER` parameters and `OBNC_REAL` for `REAL` parameters. Define a `ModuleName__Init(void)` function even if it does nothing — this is called automatically when the module is loaded and must be present.
3. **Place both files in the same directory** as the Oberon program that imports the module.

4. **Compile the client program with** `obnc`. `OBNC` detects the companion `.c` file automatically and compiles and links both files in a single step. No separate compilation of the C file is needed.

The `Delay` module illustrates all four steps: `Delay.obn` declares `Sleep` and `Flush` without bodies; `Delay.c` implements `Delay__Sleep_` and `Delay__Flush_` using `usleep` and `fflush`; both files live alongside `countDown.obn`; and `obnc countDown.obn` compiles everything in one command.

Notes on C Interfacing

Type safety at the boundary. OBNC's type checking stops at the Oberon/C boundary. The C compiler does not know about Oberon's type rules, and errors in the C code will not be caught by the Oberon compiler. Test C interface code thoroughly.

The OBNC header. Always include `<obnc/OBNC.h>` in your C file, as well as the generated module header `.obnc/ModuleName.h`. The generated header is created by OBNC during compilation and contains the declarations the C code needs to link correctly with the Oberon module.

Standard C libraries. Any standard C library can be accessed through this mechanism — `math.h`, `string.h`, `stdio.h`, or any third-party C library installed on your system.

Keep interfaces narrow. A good C interface module does one thing and presents a clean Oberon interface. The `Delay` module is a good example: two procedures, one purpose each, no exposed C types. The more C-facing code you write, the less protection Oberon's type system provides.

21.2 The extPipes Module

A **pipe** is a communication channel between two processes. One process writes to the pipe; another reads from it. The `extPipes` module allows an Oberon program to execute a shell command and either read its output or write to its input.

```
IMPORT extPipes;
```

The module provides a `Stream` type and five procedures:

```
TYPE
  Stream = POINTER TO RECORD
    eof : BOOLEAN
  END;

PROCEDURE OpenRead(command : ARRAY OF CHAR; VAR stream : Stream);
PROCEDURE OpenWrite(command : ARRAY OF CHAR; VAR stream : Stream);
PROCEDURE Read(stream : Stream; VAR ch : CHAR);
PROCEDURE Write(ch : CHAR; stream : Stream);
PROCEDURE Close(stream : Stream; VAR exitCode : INTEGER);
```

`OpenRead` executes `command` and connects `stream` to its standard output, so your program can read what the command prints, character by character.

OpenWrite executes `command` and connects `stream` to its standard input, so your program can write characters that the command receives.

Read reads one character from a read-mode stream. After reading, check `stream.eof` — it becomes `TRUE` when the command's output is exhausted.

Write writes one character to a write-mode stream.

Close closes the stream and stores the shell's exit code in `exitCode`. An exit code of `0` means success; non-zero indicates an error.

If `OpenRead` or `OpenWrite` fails, `stream` is set to `NIL`. Always check for `NIL` before proceeding.

Example 21.2: Reading from a Pipe

This program uses `OpenRead` to run the Unix `date` command and reads its output character by character, displaying the current date and time:

```
MODULE PipeRead;
  IMPORT Out, extErr, extPipes;
VAR
  stream   : extPipes.Stream;
  ch       : CHAR;
  exitCode : INTEGER;
BEGIN
  extPipes.OpenRead("date", stream);
  IF stream = NIL THEN
    extErr.String("Failed to open pipe.");
    extErr.Ln;
    ASSERT(FALSE)
  END;
  Out.String("Current date and time: ");
  extPipes.Read(stream, ch);
  WHILE ~stream.eof DO
    Out.Char(ch);
    extPipes.Read(stream, ch)
  END;
  extPipes.Close(stream, exitCode);
  IF exitCode > 0 THEN
    extErr.String("Command failed with exit code ");
    extErr.Int(exitCode, 0);
    extErr.Ln
  END
END PipeRead.
```

Output (the exact date and time will vary):

```
Current date and time: Tue 17 Jun 2025 14:32:08 BST
```

The read-loop pattern mirrors the file reading pattern from Chapter 19: read one character before the loop, then `WHILE ~stream.eof DO ... read again END`.

Example 21.3: Writing to a Pipe

This program uses `OpenWrite` to send lines of text to the Unix `sort` command, which sorts them alphabetically and prints the result to the terminal:

```
MODULE PipeWrite;
  IMPORT Out, extErr, extPipes;
CONST
  NumWords = 6;
TYPE
  Word = ARRAY 20 OF CHAR;
VAR
  words    : ARRAY NumWords OF Word;
  stream   : extPipes.Stream;
  exitCode : INTEGER;
  i, j     : INTEGER;

PROCEDURE SendLine(s : ARRAY OF CHAR; p : extPipes.Stream);
VAR
  i : INTEGER;
BEGIN
  i := 0;
  WHILE s[i] # 0X DO
    extPipes.Write(s[i], p);
    INC(i)
  END;
  extPipes.Write(0AX, p)    (* newline character: ASCII 10 *)
END SendLine;

BEGIN
  words[0] := "banana";
  words[1] := "apple";
  words[2] := "elderberry";
  words[3] := "cherry";
  words[4] := "date";
  words[5] := "fig";
  extPipes.OpenWrite("sort", stream);
  IF stream = NIL THEN
    extErr.String("Failed to open pipe.");
    extErr.Ln;
    ASSERT(FALSE)
  END;
  FOR i := 0 TO NumWords - 1 DO
    SendLine(words[i], stream)
```

```
  END;
  extPipes.Close(stream, exitCode);
  IF exitCode > 0 THEN
    extErr.String("sort failed with exit code ");
    extErr.Int(exitCode, 0);
    extErr.Ln
  END
END PipeWrite.
```

Output (printed by `sort` directly to the terminal):

```
apple
banana
cherry
date
elderberry
fig
```

The `SendLine` helper procedure writes the characters of a string one by one, followed by a newline character (`0AX` — ASCII 10). Each `extPipes.Write` call sends one character to the `sort` command's standard input. When `Close` is called, the pipe is closed and `sort` receives the end-of-file signal, at which point it sorts the lines it has received and prints them.

When to Use Pipes

Pipes are useful when you want to leverage existing command-line tools from within an Oberon program — sorting, text transformation, querying system information. They are not appropriate for performance-critical work (the overhead of spawning a shell process is significant) but for simple integration with Unix tools they are clean and effective.

21.3 Graphics with XYplane

The `XYplane` module provides basic pixel-level graphics in a window. It is simple by design — a rectangular drawing surface where individual pixels can be turned on or off and colours can be set — but from these primitives you can build curves, shapes, and interactive interfaces.

```
IMPORT XYplane;
```

The Drawing Plane

Before drawing anything, the window must be initialised:

```
XYplane.Open
```

To set a specific size before opening, call `XYplane.SetSize(width, height)` first:

```
XYplane.SetSize(400, 300);
XYplane.Open
```

After `Open`, the variables `XYplane.W` and `XYplane.H` hold the actual width and height of the window in pixels.

Key Procedures

`XYplane.Dot(x, y, mode)` draws or erases the pixel at `(x, y)`, where `(0, 0)` is the **lower-left corner** of the window and `y` increases upwards. `mode` is `XYplane.draw` (1) to draw or `XYplane.erase` (0) to erase.

`XYplane.UseColor(color)` sets the drawing colour as a 24-bit RGB integer: `0FF0000H` = red, `000FF00H` = green, `00000FFH` = blue, `0FFFFFFH` = white.

`XYplane.Key()` reads the keyboard and returns the character of any key pressed, or `0X` if none. Crucially, it also **repaints the window** — drawing commands only become visible when `Key` is called. The standard event loop is therefore `REPEAT ... UNTIL XYplane.Key() = "q"`.

`XYplane.Clear` erases all pixels in the window.

`XYplane.IsDot(x, y)` returns `TRUE` if the pixel at `(x, y)` is drawn.

Example 21.4: Plotting x²

This program plots the curve `y = x²` scaled to fit the drawing plane:

```
MODULE PlotSquare;
  IMPORT XYplane;
VAR
  x, y     : INTEGER;
  xf, yf   : REAL;
  xScale   : REAL;
  yScale   : REAL;
BEGIN
  XYplane.SetSize(400, 400);
  XYplane.Open;
  XYplane.UseColor(000FF00H);    (* green *)
  (* Scale so x runs from -10 to 10 across the width,
     and y runs from 0 to 100 up the height *)
  xScale := FLT(XYplane.W) / 20.0;   (* pixels per unit x *)
  yScale := FLT(XYplane.H) / 100.0;  (* pixels per unit y *)
  FOR x := 0 TO XYplane.W - 1 DO
    xf := FLT(x) / xScale - 10.0;   (* x in range -10 to 10 *)
    yf := xf * xf;                   (* y = x^2, range 0 to 100 *)
    y  := FLOOR(yf * yScale);
    IF (y >= 0) & (y < XYplane.H) THEN
      XYplane.Dot(x, y, XYplane.draw)
```

```
    END
  END;
  REPEAT UNTIL XYplane.Key() = "q"
END PlotSquare.
```

The curve runs from `x = -10` to `x = 10` horizontally. For each pixel column `x`, the corresponding mathematical x-value is computed, `y = x²` is calculated, and the result is scaled to a pixel row. The `IF` guard prevents out-of-bounds drawing for any values that fall outside the window.

21.4 Extended Graphics: RectUI and XYtext

`XYplane` gives you pixels. Two companion modules — `RectUI` and `XYtext` — build on top of it to provide rectangular buttons with mouse interaction, and text rendering in the graphics window. These modules are available from the download link given in the preface; place `RectUI.obn` and `XYtext.obn` in the same directory as your program. Note that in order to run the `ButtonDemo` program below you will also need `Delay.obn` and `Delay.c` from section 21.1.

The RectUI Module

`RectUI` manages a set of rectangular buttons and handles mouse events using a simple polling model. The typical interaction sequence is:

1. Call `RectUI.Init` to open the window and initialise the button list.
2. Add buttons with `RectUI.AddButton`.
3. Optionally set button colours with `RectUI.SetColor`.
4. Call `RectUI.Draw` to render all buttons.
5. Repeatedly call `RectUI.Poll` in an event loop to check for clicks.

```
IMPORT RectUI;
```

The exported procedures are:

`RectUI.Init(width, height : INTEGER)` — opens the graphics window and clears the internal button list.

`RectUI.AddButton(x, y, w, h : INTEGER; label : ARRAY OF CHAR; scale : INTEGER; VAR id : RectUI.RectId)` — creates a button with its lower-left corner at `(x, y)`, width `w`, height `h`, and the given text label. `scale` is the text size multiplier (1 = normal). The button's identifier is returned in `id` and is used in subsequent calls.

`RectUI.SetColor(id : RectUI.RectId; color : INTEGER)` — sets the outline colour of button `id` to the given 24-bit RGB colour.

`RectUI.Draw` — draws all buttons and their labels in the window.

`RectUI.Poll(VAR id : RectUI.RectId)` — checks for a mouse click. If a button was clicked, its identifier is returned in `id`. If no click occurred, `id` is set to `-1`. `Poll` also calls `Delay.Sleep(50)` internally to avoid busy-waiting.

`RectUI.ClearRect(x, y, w, h : INTEGER)` — erases a rectangular region of the window. Useful for

clearing a display area before redrawing updated content.

The XYtext Module

`XYtext` renders text into the graphics window using a built-in 8×8 bitmap font.

```
IMPORT XYtext;
```

`XYtext.DrawChar(x, y : INTEGER; c : CHAR; scale : INTEGER)` — draws the single character `c` at pixel position `(x, y)` with the given scale factor.

`XYtext.DrawString(x, y : INTEGER; s : ARRAY OF CHAR; scale : INTEGER)` — draws the string `s` starting at pixel position `(x, y)`.

`XYtext.CenterString(x, y, w, h : INTEGER; s : ARRAY OF CHAR; scale : INTEGER)` — draws `s` centred within the rectangle at `(x, y)` with width `w` and height `h`. This is used internally by `RectUI` to centre button labels.

All text is drawn in the current `XYplane` drawing colour, set with `XYplane.UseColor`. The `scale` parameter multiplies the character size: `scale = 1` gives 8×8 pixel characters, `scale = 2` gives 16×16, and so on.

Example 21.5: ButtonDemo

The following program creates three coloured buttons labelled SMALL, MEDIUM, and LARGE. Clicking a button increments a corresponding counter displayed above the buttons. A QUIT button exits the program. The counters are displayed as strings using `extConvert.IntToString`.

Note that `extConvert.IntToString` requires its `signed` parameter to be a variable, not a literal — this is why `signed` is declared as a `BOOLEAN` variable and set to `TRUE` before use.

```
MODULE ButtonDemo;
  IMPORT RectUI, XYtext, extConvert;
CONST
  red    = 0FF0000H;
  yellow = 0FFFF00H;
  white  = 0FFFFFFH;
  green  = 000FF00H;
VAR
  smallID,
  mediumID,
  largeID,
  quitID       : RectUI.RectId;
  id           : RectUI.RectId;
  smallCount,
  mediumCount,
  largeCount   : INTEGER;
  s            : ARRAY 16 OF CHAR;
  signed       : BOOLEAN;
```

```
PROCEDURE DrawCounters;
BEGIN
  RectUI.ClearRect(20, 120, 250, 40);
  XYtext.DrawString(20, 140, "SMALL:", 1);
  extConvert.IntToString(smallCount, s, signed);
  XYtext.DrawString(80, 140, s, 1);
  XYtext.DrawString(20, 130, "MEDIUM:", 1);
  extConvert.IntToString(mediumCount, s, signed);
  XYtext.DrawString(90, 130, s, 1);
  XYtext.DrawString(20, 120, "LARGE:", 1);
  extConvert.IntToString(largeCount, s, signed);
  XYtext.DrawString(80, 120, s, 1)
END DrawCounters;

BEGIN
  signed := TRUE;
  RectUI.Init(400, 200);
  RectUI.AddButton(20,  40, 80, 40, "SMALL", 1, smallID);
  RectUI.SetColor(smallID,  green);
  RectUI.AddButton(120, 40, 80, 40, "MEDIUM", 1, mediumID);
  RectUI.SetColor(mediumID, yellow);
  RectUI.AddButton(220, 40, 80, 40, "LARGE", 1, largeID);
  RectUI.SetColor(largeID,  white);
  RectUI.AddButton(320, 10, 60, 30, "QUIT", 1, quitID);
  RectUI.SetColor(quitID,   red);
  RectUI.Draw;
  DrawCounters;
  REPEAT
    RectUI.Poll(id);
    IF id = smallID THEN
      INC(smallCount);
      DrawCounters
    ELSIF id = mediumID THEN
      INC(mediumCount);
      DrawCounters
    ELSIF id = largeID THEN
      INC(largeCount);
      DrawCounters
    END
  UNTIL id = quitID
END ButtonDemo.
```

The `DrawCounters` procedure clears the counter area with `RectUI.ClearRect` before redrawing the updated values — this overwrites the old numbers rather than drawing on top of them. The event loop runs until the QUIT button is clicked, at which point `id = quitID` and the `REPEAT…UNTIL` exits.

Summary

- **C interfacing** in OBNC works by placing a companion `.c` file alongside the `.obn` file. Procedures declared without a body in Oberon are implemented in C, following the `Module__Procedure_` naming convention. Include `<obnc/OBNC.h>` and the generated module header `.obnc/ModuleName.h` in the C file. Always define a `ModuleName__Init(void)` function. The `Delay` module demonstrates this: `Delay.Sleep` calls `usleep` for millisecond-precision pauses; `Delay.Flush` calls `fflush(stdout)` to force buffered output to appear immediately.
- Output on the same line requires `Delay.Flush` after each value to prevent buffering from holding back the display. Output followed by `Out.Ln` does not need flushing, as the newline flushes the buffer automatically.
- The **`extPipes`** module executes shell commands and connects them to the Oberon program via a pipe. `OpenRead` captures a command's standard output; `OpenWrite` sends data to a command's standard input. Read and write one character at a time; check `stream.eof` to detect end of output; always call `Close` when done.
- The **`XYplane`** module provides pixel-level graphics. Call `XYplane.Open` (optionally after `XYplane.SetSize`) to initialise the window. Draw pixels with `XYplane.Dot(x, y, mode)`, where `(0, 0)` is the lower-left corner and `y` increases upwards. `XYplane.Key()` both reads the keyboard and repaints the window.
- **`RectUI`** adds rectangular buttons with mouse polling: `Init`, `AddButton`, `SetColor`, `Draw`, `Poll`, and `ClearRect`. **`XYtext`** adds text rendering: `DrawChar`, `DrawString`, and `CenterString`. Both modules require `Delay` as a dependency.

Exercises

21.1 Modify the `countDown` program to count down on separate lines (one number per line) and remove the `Delay.Flush` call. Verify that the output still appears correctly one number at a time. Then restore the single-line version and remove just the `Delay.Flush` call — observe what happens.

21.2 Write a program using `extPipes.OpenRead` to run the command `uname -a` and display the output. This command prints information about the operating system and hardware.

21.3 Write a program using `extPipes.OpenWrite` to send five lines of text to the `wc -l` command (which counts lines) and display the result. Since `wc` prints its result to the terminal after the pipe is closed, you will see the count appear when `Close` is called.

21.4 Modify the `PlotSquare` program to plot the curve `y = sin(x)` across the width of the window, for `x` from 0 to 2π. Scale the y-axis so the curve fills the height of the window. Use a different colour from the `x²` example.

21.5 Write a program using `XYplane` that draws a grid of horizontal and vertical lines, dividing the window into equal cells. Define the cell size as a constant. Use `XYplane.UseColor` to draw the grid in a colour of your choice.

21.6 Extend the `ButtonDemo` program with a RESET button that sets all three counters back to zero and redraws them. Add it in a suitable position in the window and give it a distinct colour.

21.7 Write a C interface module `Clock` that exports a function `Clock.Ticks() : INTEGER` implemented using the C `clock()` function from `<time.h>`, which returns the number of processor

clock ticks since the program started. Write an Oberon program that uses it to measure how long a loop of one million iterations takes to execute.

Part V — Putting It All Together

- Chapter 22 — Object-Oriented Ideas
- Chapter 23 — A Text-Based Address Book

Chapter 22 — Object-Oriented Ideas

Most of the programs in this book have been organised around procedures and data structures: procedures that operate on records, arrays of records, linked lists, and so on. This style of programming works well for many problems, but it has a limitation. When a program needs to handle several related but distinct kinds of thing — different shapes, different file formats, different kinds of user interface element — the code that processes them tends to become cluttered with `IF/ELSIF` chains that test what kind of thing is being dealt with and then branch accordingly. Adding a new kind of thing means finding and updating every one of those chains.

Object-oriented programming is an approach that addresses this problem by organising code differently; it is set of techniques for building flexible, extensible systems without requiring any additional language features beyond those we've already learned.

Rather than keeping data and the procedures that act on it as separate concerns, the two are brought together: each kind of thing carries not only its data but also the procedures that define how it behaves. Instead of asking "what kind of thing is this, and what should I do with it?", the program simply asks the thing itself to perform the operation. Each kind of thing knows how to respond in its own way, and the code that drives the program does not need to know which specific kind it is dealing with. The result is that adding a new kind of thing requires only writing its own data and behaviour — the rest of the program needs no changes. Oberon-07 achieves this style of programming through three mechanisms: record type extension, which allows a record type to be built on top of an existing one; pointers, which allow collections to contain different extended types; and procedure types in records, which allow each record to carry its own associated procedures. The first of these is new to this chapter; the other two are already familiar.

This is one of the more advanced chapters in the book. Do not be discouraged if some of the ideas take time to absorb — they build on records, pointers, and procedure types from earlier chapters, and are best understood by working through the examples carefully.

22.1 The Problem: Heterogeneous Collections

Consider a simple drawing program. It might work with several kinds of shape — circles, rectangles, lines, triangles. Each shape has some common properties (a position, a colour) and some properties specific to its kind (a radius for a circle, width and height for a rectangle).

With what we have learned so far, we might declare a separate record type for each shape. But then we cannot store them in a single array — an array of `CircleDesc` records cannot hold a `RectangleDesc`. We would need a separate array for each shape type, and a separate loop to process each.

This is awkward and inflexible. Adding a new shape type requires modifying every procedure that processes shapes. What we want is a single collection that can hold shapes of different types — a **heterogeneous** collection — and a way to process each element according to its actual type.

Oberon-07's answer is **type extension**.

22.2 Record Extension

A record type can be declared as an **extension** of another record type. The extended type inherits all

the fields of its **base type** and adds new fields of its own:

```
TYPE
  BaseType = RECORD
    field1 : type1;
    field2 : type2
  END;

  ExtendedType = RECORD (BaseType)
    field3 : type3;
    field4 : type4
  END;
```

The notation `RECORD (BaseType)` declares `ExtendedType` as an extension of `BaseType`. A variable of type `ExtendedType` contains all the fields of `BaseType` (`field1`, `field2`) plus its own new fields (`field3`, `field4`).

This is sometimes described as **inheritance** in object-oriented terminology, though Wirth himself prefers not to use anthropomorphic terms. We will say simply that the extended type **extends** the base type with additional fields.

A Concrete Example

```
TYPE
  Shape = RECORD
    x, y  : INTEGER;    (* position *)
    colour : INTEGER     (* RGB colour *)
  END;

  Circle = RECORD (Shape)
    radius : INTEGER
  END;

  Rectangle = RECORD (Shape)
    width, height : INTEGER
  END;
```

A `Circle` variable contains `x`, `y`, `colour`, and `radius`. A `Rectangle` variable contains `x`, `y`, `colour`, `width`, and `height`. Both also contain everything that `Shape` contains.

Extension and Pointers

Type extension becomes truly powerful when combined with pointers. Because a pointer to an extended type is **compatible** with a pointer to its base type, a linked list declared through the base type can hold nodes of any extended type:

```
TYPE
  ShapePtr = POINTER TO Shape;
  CirclePtr = POINTER TO Circle;
  RectPtr   = POINTER TO Rectangle;

  Shape = RECORD
    x, y   : INTEGER;
    colour  : INTEGER;
    next    : ShapePtr    (* linked list of shapes *)
  END;

  Circle = RECORD (Shape)
    radius : INTEGER
  END;

  Rectangle = RECORD (Shape)
    width, height : INTEGER
  END;
```

We can now build a list that mixes circles and rectangles:

```
VAR
  list : ShapePtr;
  c    : CirclePtr;
  r    : RectPtr;
...
NEW(c);
c.x := 10; c.y := 20; c.radius := 50;
c.next := list;
list   := c;

NEW(r);
r.x := 100; r.y := 50; r.width := 80; r.height := 40;
r.next := list;
list   := r;
```

The list variable is of type `ShapePtr`, but it can hold pointers to any extension of `Shape`. This is the key to heterogeneous collections in Oberon-07.

22.3 The IS Type Test

When traversing a heterogeneous collection, we often need to know the actual type of each element — is this a `Circle` or a `Rectangle`? Oberon-07 provides the **IS** operator for this purpose:

```
variable IS TypeName
```

This is a Boolean expression that is `TRUE` if `variable` currently holds a value of type `TypeName` (or an extension of it), and `FALSE` otherwise.

```
VAR p : ShapePtr;
...
IF p IS CirclePtr THEN
  Out.String("This is a circle")
ELSIF p IS RectPtr THEN
  Out.String("This is a rectangle")
END
```

The `IS` operator can be applied to pointer variables and to `VAR` parameters of record type. The type being tested must be an extension of the variable's declared type.

22.4 Type Guards

Once you have determined that a pointer `p` is of type `CirclePtr`, you need to access the circle-specific fields (`radius`). But `p` is declared as `ShapePtr` — the compiler will not let you write `p.radius` because `Shape` has no such field.

The solution is a **type guard**: an assertion that a pointer is of a specific extended type, written as `p(TypeName)`:

```
VAR p : ShapePtr;
...
IF p IS CirclePtr THEN
  Out.Int(p(CirclePtr).radius, 0)
END
```

`p(CirclePtr)` asserts that `p` is a `CirclePtr` and lets you access the `CirclePtr` fields. If at runtime `p` is not actually a `CirclePtr`, the program traps with a type guard failure (exception code 6 from Chapter 20).

The `IS` test and the type guard are complementary: use `IS` to check the type first, then use the type guard to access type-specific fields. Writing the type guard without the `IS` check is fine if you are certain of the type from context — the runtime check provides safety either way.

Example 22.1: A Mixed Shape List

This program builds a list of circles and rectangles, then traverses it and prints information about each shape:

```
MODULE ShapeList;
  IMPORT Out;
TYPE
  ShapePtr = POINTER TO Shape;
  CirclePtr = POINTER TO Circle;
```

```
  RectPtr   = POINTER TO Rectangle;

  Shape = RECORD
    x, y  : INTEGER;
    next  : ShapePtr
  END;

  Circle = RECORD (Shape)
    radius : INTEGER
  END;

  Rectangle = RECORD (Shape)
    width, height : INTEGER
  END;

VAR
  list : ShapePtr;
  c    : CirclePtr;
  r    : RectPtr;
  p    : ShapePtr;

BEGIN
  (* Build list: circle, rectangle, circle *)
  NEW(c); c.x := 10; c.y := 20; c.radius := 50; c.next := NIL;
  list := c;
  NEW(r); r.x := 100; r.y := 50; r.width := 80; r.height := 40;
  r.next := list; list := r;
  NEW(c); c.x := 200; c.y := 150; c.radius := 30;
  c.next := list; list := c;
  (* Traverse and describe each shape *)
  p := list;
  WHILE p # NIL DO
    Out.String("Shape at (");
    Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
    Out.String("): ");
    IF p IS CirclePtr THEN
      Out.String("circle, radius = ");
      Out.Int(p(CirclePtr).radius, 0)
    ELSIF p IS RectPtr THEN
      Out.String("rectangle, ");
      Out.Int(p(RectPtr).width, 0);
      Out.String(" x ");
      Out.Int(p(RectPtr).height, 0)
    END;
    Out.Ln;
    p := p.next
```

```
  END
END ShapeList.
```

Output:

```
Shape at (200, 150): circle, radius = 30
Shape at (100, 50): rectangle, 80 x 40
Shape at (10, 20): circle, radius = 50
```

The list is traversed through the `ShapePtr` variable `p`. At each node, `IS` determines the actual type and the type guard accesses the type-specific fields. The common fields `x`, `y`, and `next` are accessible directly through `p` without any type guard, because they belong to the base type `Shape`.

22.5 Procedure Fields in Records

Before exploring the three extension patterns, it is worth noting that records can contain fields of **procedure type**, just as they can contain fields of any other type. This combines naturally with type extension and gives a powerful way to associate different behaviour with different extended types.

Recall from Chapter 14 that a procedure type describes the signature of a procedure:

```
TYPE
  DrawProc = PROCEDURE(x, y : INTEGER);
```

A record field of this type can hold any procedure matching that signature:

```
TYPE
  ShapeDesc = RECORD
    x, y : INTEGER;
    draw : DrawProc
  END;
```

When a `ShapeDesc` variable is created, a specific procedure is assigned to its `draw` field. Calling `shape.draw(...)` then calls whatever procedure is stored there — which may be different for different shapes. The procedure stored in the field is chosen at runtime rather than fixed at compile time.

This is the key idea that the three extension patterns in the next section build upon. The patterns differ only in how and where the procedure fields are stored.

22.6 Three Extension Patterns

Landström's *Data Abstraction in Oberon* documentation (included in the OBNC package) describes three distinct patterns for associating procedures with extended record types. Each is a different arrangement of type extension and procedure fields. We will illustrate all three using a simple shape hierarchy.

Pattern 1: Procedure Fields in Each Record

The simplest pattern adds a procedure field directly to the base record type. When a new shape is created, the appropriate procedure is assigned to that field. The base record also contains a `next` pointer so shapes can be linked into a list. The whole thing — type declarations, procedures, data, and the main block — fits in a single self-contained program:

```
MODULE Pattern1;
  IMPORT Out;
TYPE
  ShapePtr  = POINTER TO ShapeDesc;
  CirclePtr = POINTER TO CircleDesc;
  RectPtr   = POINTER TO RectDesc;
  DrawProc  = PROCEDURE(p : ShapePtr);

  ShapeDesc = RECORD
    x, y : INTEGER;
    draw : DrawProc;
    next : ShapePtr
  END;

  CircleDesc = RECORD (ShapeDesc)
    radius : INTEGER
  END;

  RectDesc = RECORD (ShapeDesc)
    width, height : INTEGER
  END;

VAR
  list, p : ShapePtr;
  c       : CirclePtr;
  r       : RectPtr;

PROCEDURE DrawCircle(p : ShapePtr);
BEGIN
  Out.String("Circle at (");
  Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
  Out.String("), radius = ");
  Out.Int(p(CirclePtr).radius, 0);
  Out.Ln
END DrawCircle;

PROCEDURE DrawRect(p : ShapePtr);
BEGIN
  Out.String("Rectangle at (");
  Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
```

```
  Out.String("), ");
  Out.Int(p(RectPtr).width, 0);
  Out.String(" x ");
  Out.Int(p(RectPtr).height, 0);
  Out.Ln
END DrawRect;

BEGIN
  list := NIL;
  NEW(c); c.x := 10;  c.y := 20;  c.radius := 50;
  c.draw := DrawCircle; c.next := list; list := c;
  NEW(r); r.x := 100; r.y := 50;  r.width := 80; r.height := 40;
  r.draw := DrawRect;   r.next := list; list := r;
  NEW(c); c.x := 200; c.y := 150; c.radius := 30;
  c.draw := DrawCircle; c.next := list; list := c;
  Out.String("All shapes (Pattern 1):"); Out.Ln;
  p := list;
  WHILE p # NIL DO
    p.draw(p);
    p := p.next
  END
END Pattern1.
```

Output:

```
All shapes (Pattern 1):
Circle at (200, 150), radius = 30
Rectangle at (100, 50), 80 x 40
Circle at (10, 20), radius = 50
```

The call `p.draw(p)` always invokes the right procedure for the actual shape, with no `IS` test or `IF`/`ELSIF` chain needed. The shape is passed as an explicit parameter — Oberon-07 has no hidden mechanism for this, which keeps the behaviour transparent.

The main drawback is that every shape record carries its own copy of every procedure field. For a small number of procedure fields and shapes this is fine, but it becomes wasteful if there are many of each.

Pattern 2: Shared Procedure Table

When there are many procedure fields and many records, it is more memory-efficient to store the procedure fields in a single **shared record** and give each shape a pointer to it. All circles share one procedure table record; all rectangles share another. Only two table records exist regardless of how many shapes are created.

```
MODULE Pattern2;
  IMPORT Out;
TYPE
  ShapePtr  = POINTER TO ShapeDesc;
  CirclePtr = POINTER TO CircleDesc;
  RectPtr   = POINTER TO RectDesc;
  DrawProc  = PROCEDURE(p : ShapePtr);

  ProcsDesc = RECORD
    draw : DrawProc
  END;
  ProcsPtr = POINTER TO ProcsDesc;

  ShapeDesc = RECORD
    x, y : INTEGER;
    do   : ProcsPtr;
    next : ShapePtr
  END;

  CircleDesc = RECORD (ShapeDesc)
    radius : INTEGER
  END;

  RectDesc = RECORD (ShapeDesc)
    width, height : INTEGER
  END;

VAR
  list, p      : ShapePtr;
  c            : CirclePtr;
  r            : RectPtr;
  circleProcs  : ProcsPtr;
  rectProcs    : ProcsPtr;

PROCEDURE DrawCircle(p : ShapePtr);
BEGIN
  Out.String("Circle at (");
  Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
  Out.String("), radius = ");
  Out.Int(p(CirclePtr).radius, 0);
  Out.Ln
END DrawCircle;

PROCEDURE DrawRect(p : ShapePtr);
BEGIN
  Out.String("Rectangle at (");
```

```
  Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
  Out.String("), ");
  Out.Int(p(RectPtr).width, 0);
  Out.String(" x ");
  Out.Int(p(RectPtr).height, 0);
  Out.Ln
END DrawRect;

BEGIN
  (* Create the shared procedure table records once *)
  NEW(circleProcs); circleProcs.draw := DrawCircle;
  NEW(rectProcs);   rectProcs.draw   := DrawRect;
  list := NIL;
  NEW(c); c.x := 10;  c.y := 20;  c.radius := 50;
  c.do := circleProcs; c.next := list; list := c;
  NEW(r); r.x := 100; r.y := 50;  r.width := 80; r.height := 40;
  r.do := rectProcs;   r.next := list; list := r;
  NEW(c); c.x := 200; c.y := 150; c.radius := 30;
  c.do := circleProcs; c.next := list; list := c;
  Out.String("All shapes (Pattern 2):"); Out.Ln;
  p := list;
  WHILE p # NIL DO
    p.do.draw(p);
    p := p.next
  END
END Pattern2.
```

Output:

```
All shapes (Pattern 2):
Circle at (200, 150), radius = 30
Rectangle at (100, 50), 80 x 40
Circle at (10, 20), radius = 50
```

The output is identical to Pattern 1. The difference is entirely internal: instead of each record carrying its own copy of `DrawCircle` or `DrawRect`, every circle points to `circleProcs` and every rectangle to `rectProcs`. The call `p.do.draw(p)` reaches the procedure through that shared pointer.

Pattern 3: A Single Handler with Extended Request Records

The third pattern takes a different approach. Instead of separate procedure fields for each operation, the shape record has a single **handler** procedure that accepts a **request record** parameter. The type of the request record identifies which operation is being requested. Because request records are themselves extensible, new operations can be added without modifying the existing shape types at all.

```
MODULE Pattern3;
  IMPORT Out;
TYPE
  ShapePtr  = POINTER TO ShapeDesc;
  CirclePtr = POINTER TO CircleDesc;
  RectPtr   = POINTER TO RectDesc;
  ReqPtr    = POINTER TO ReqDesc;
  DrawReqPtr = POINTER TO DrawReq;
  MoveReqPtr = POINTER TO MoveReq;
  Handler   = PROCEDURE(p : ShapePtr; req : ReqPtr);

  ReqDesc  = RECORD END;           (* base request type — extended per operation *)
  DrawReq  = RECORD (ReqDesc) END; (* a request to draw                          *)
  MoveReq  = RECORD (ReqDesc)      (* a request to move by (dx, dy)              *)
    dx, dy : INTEGER
  END;

  ShapeDesc = RECORD
    x, y   : INTEGER;
    handle : Handler;
    next   : ShapePtr
  END;

  CircleDesc = RECORD (ShapeDesc)
    radius : INTEGER
  END;

  RectDesc = RECORD (ShapeDesc)
    width, height : INTEGER
  END;

VAR
  list, p  : ShapePtr;
  c        : CirclePtr;
  r        : RectPtr;
  drawReq  : DrawReqPtr;
  moveReq  : MoveReqPtr;

PROCEDURE HandleCircle(p : ShapePtr; req : ReqPtr);
BEGIN
  IF req IS DrawReqPtr THEN
    Out.String("Circle at (");
    Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
    Out.String("), radius = ");
    Out.Int(p(CirclePtr).radius, 0);
    Out.Ln
```

```
  ELSIF req IS MoveReqPtr THEN
    INC(p.x, req(MoveReqPtr).dx);
    INC(p.y, req(MoveReqPtr).dy)
  END
  (* unknown requests are silently ignored *)
END HandleCircle;

PROCEDURE HandleRect(p : ShapePtr; req : ReqPtr);
BEGIN
  IF req IS DrawReqPtr THEN
    Out.String("Rectangle at (");
    Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
    Out.String("), ");
    Out.Int(p(RectPtr).width, 0);
    Out.String(" x ");
    Out.Int(p(RectPtr).height, 0);
    Out.Ln
  ELSIF req IS MoveReqPtr THEN
    INC(p.x, req(MoveReqPtr).dx);
    INC(p.y, req(MoveReqPtr).dy)
  END
END HandleRect;

PROCEDURE SendToAll(req : ReqPtr);
BEGIN
  p := list;
  WHILE p # NIL DO
    p.handle(p, req);
    p := p.next
  END
END SendToAll;

BEGIN
  list := NIL;
  NEW(c); c.x := 10;  c.y := 20;  c.radius := 50;
  c.handle := HandleCircle; c.next := list; list := c;
  NEW(r); r.x := 100; r.y := 50;  r.width := 80; r.height := 40;
  r.handle := HandleRect;   r.next := list; list := r;
  NEW(c); c.x := 200; c.y := 150; c.radius := 30;
  c.handle := HandleCircle; c.next := list; list := c;
  NEW(drawReq);
  Out.String("All shapes (Pattern 3):"); Out.Ln;
  SendToAll(drawReq);
  NEW(moveReq); moveReq.dx := 5; moveReq.dy := 10;
  SendToAll(moveReq);
  Out.Ln;
```

```
  Out.String("After moving by (5, 10):"); Out.Ln;
  SendToAll(drawReq)
END Pattern3.
```

Output:

```
All shapes (Pattern 3):
Circle at (200, 150), radius = 30
Rectangle at (100, 50), 80 x 40
Circle at (10, 20), radius = 50

After moving by (5, 10):
Circle at (205, 160), radius = 30
Rectangle at (105, 60), 80 x 40
Circle at (15, 30), radius = 50
```

`SendToAll(drawReq)` sends a draw request to every shape in the list; `SendToAll(moveReq)` sends a move request. Each handler responds to the requests it recognises and silently ignores the rest. `SendToAll` itself is completely unaware of how many shape types exist — it simply passes the request along the list. When a new shape type is added, only its handler procedure needs to be written; `SendToAll` requires no changes.

Comparing the Three Patterns

The three patterns involve the same ingredients — type extension, pointers, and procedure fields — arranged differently:

In **Pattern 1**, each record carries its own copy of every procedure field. This is the simplest arrangement and is appropriate when there are few procedure fields or few records.

In **Pattern 2**, procedure fields are stored once in a shared record, and each shape carries a pointer to the appropriate shared record. This uses less memory when there are many records with the same set of procedures.

In **Pattern 3**, each shape has a single handler procedure that dispatches based on the type of an extended request record. This is the most flexible arrangement: new request types can be defined in new modules without touching the shape types, and a single `SendToAll` procedure works for any request.

22.7 Example: A Mixed Shape Collection

Let us bring everything together in a program that uses type extension, `IS`, type guards, and procedure fields to manage a collection of mixed shapes. We use Pattern 1 — a procedure field in each record — to keep the example focused.

```
MODULE Shapes;
  IMPORT Out;
```

```
TYPE
  ShapePtr  = POINTER TO ShapeDesc;
  CirclePtr = POINTER TO CircleDesc;
  RectPtr   = POINTER TO RectDesc;
  DrawProc  = PROCEDURE(p : ShapePtr);

  ShapeDesc = RECORD
    x, y : INTEGER;
    draw : DrawProc;
    next : ShapePtr
  END;

  CircleDesc = RECORD (ShapeDesc)
    radius : INTEGER
  END;

  RectDesc = RECORD (ShapeDesc)
    width, height : INTEGER
  END;

VAR
  list : ShapePtr;
  c    : CirclePtr;
  r    : RectPtr;
  p    : ShapePtr;

PROCEDURE DrawCircle(p : ShapePtr);
BEGIN
  Out.String("Circle at (");
  Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
  Out.String("), radius = ");
  Out.Int(p(CirclePtr).radius, 0);
  Out.Ln
END DrawCircle;

PROCEDURE DrawRect(p : ShapePtr);
BEGIN
  Out.String("Rectangle at (");
  Out.Int(p.x, 0); Out.String(", "); Out.Int(p.y, 0);
  Out.String("), ");
  Out.Int(p(RectPtr).width, 0);
  Out.String(" x ");
  Out.Int(p(RectPtr).height, 0);
  Out.Ln
END DrawRect;
```

```
PROCEDURE AddCircle(x, y, radius : INTEGER);
BEGIN
  NEW(c);
  c.x      := x;
  c.y      := y;
  c.radius := radius;
  c.draw   := DrawCircle;
  c.next   := list;
  list     := c
END AddCircle;

PROCEDURE AddRect(x, y, width, height : INTEGER);
BEGIN
  NEW(r);
  r.x      := x;
  r.y      := y;
  r.width  := width;
  r.height := height;
  r.draw   := DrawRect;
  r.next   := list;
  list     := r
END AddRect;

PROCEDURE DrawAll;
BEGIN
  p := list;
  WHILE p # NIL DO
    p.draw(p);
    p := p.next
  END
END DrawAll;

BEGIN
  list := NIL;
  AddCircle(10, 20, 50);
  AddRect(100, 50, 80, 40);
  AddCircle(200, 150, 30);
  AddRect(300, 200, 60, 60);
  Out.String("All shapes:");
  Out.Ln;
  DrawAll
END Shapes.
```

Output:

```
All shapes:
Rectangle at (300, 200), 60 x 60
Circle at (200, 150), radius = 30
Rectangle at (100, 50), 80 x 40
Circle at (10, 20), radius = 50
```

The shapes appear in reverse order of insertion because each new shape is prepended to the list. The `DrawAll` procedure is elegantly simple: it traverses the list and calls `p.draw(p)` for each shape — the right drawing procedure is called automatically for each shape type, with no `IF/IS` chain needed. Adding a new shape type (a triangle, say) requires writing `DrawTriangle` and an `AddTriangle` procedure; `DrawAll` needs no change at all.

Summary

- **Type extension** allows a record type to be declared as an extension of a base type: `RECORD (BaseType) ... END`. The extended type gains all the fields of the base type and adds new ones of its own.
- A pointer to an extended type is compatible with a pointer to its base type. This allows a linked list or other collection to hold records of different extended types — a **heterogeneous collection**.
- The **IS** operator (`p IS T`) tests whether pointer `p` currently holds a value of type `T` (or an extension of it). It returns a `BOOLEAN`.
- A **type guard** (`p(T)`) asserts that pointer `p` is of type `T` and grants access to `T`'s fields. If the assertion fails at runtime, the program traps with exception code 6.
- Records can contain fields of **procedure type**. Assigning a specific procedure to such a field at the time a record is created allows different records of extended types to carry different behaviour in the same field.
- Three patterns for associating procedures with extended record types are described in Landström's *Data Abstraction in Oberon*:
    - **Pattern 1**: a procedure field per operation in each record. Simple; appropriate for small numbers of procedure fields and records.
    - **Pattern 2**: procedure fields stored once in a shared record; each shape carries a pointer to the appropriate shared record. More memory-efficient when there are many records of the same extended type.
    - **Pattern 3**: a single handler procedure per record, accepting an extended request record. The most flexible arrangement: new request types can be added in new modules without modifying existing types, and a single traversal procedure works for any request.

Exercises

22.1 Extend the `ShapeList` program from Section 22.4 to include a `TriangleDesc` type with fields `base` and `height`. Add several triangles to the list alongside circles and rectangles, and print appropriate output for each type during traversal.

22.2 Add a `move` procedure field to the `ShapeDesc` base record in the worked example from Section 22.7. Write `MoveCircle` and `MoveRect` procedures that update `x` and `y`, assign them during record creation, and write a `MoveAll(dx, dy : INTEGER)` procedure that applies the move to every shape in the list.

22.3 Rewrite the worked example from Section 22.7 using Pattern 3 — a single handler with extended request records. Define `DrawReq` and `MoveReq` request types. Replace the `draw` field with a single `handle` field. Write a `SendToAll` procedure and test it with both request types.

22.4 Write a program that uses type extension to build a simple expression tree. Define a base pointer type and three extended record types: one holding an integer value, one representing addition (with two child pointers), and one representing multiplication (also with two child pointers). Write an `Eval` function procedure that traverses the tree using `IS` and type guards to compute the result. Build and evaluate the expression `(3 + 4) * (2 + 5)`.

22.5 Using Pattern 2 from Section 22.6, add a `TriangleDesc` extended type with its own shared procedure record. Verify that all three shape types can be stored in the same list and that calling `p.do.draw(p)` calls the correct procedure for each.

22.6 Using Pattern 3, add a new request type `AreaReq` with a field `result : REAL`. Write handlers for circles (area = π × radius²) and rectangles (area = width × height) that fill in `result` when they receive an `AreaReq`. Write a `SendToAll` procedure that sends an `AreaReq` to every shape in the list and prints the total area. Verify that shapes whose handlers do not recognise `AreaReq` leave `result` unchanged.

Chapter 23 — A Text-Based Address Book

The programs we have written so far have been self-contained and relatively small — designed to illustrate one concept at a time. Real programs are larger, involve multiple source files, and require sustained design effort before a line of code is written. This chapter walks through the design and implementation of a complete, useful program: a text-based address book.

The address book will store contacts — names, phone numbers, and email addresses — in a file that persists between runs. The user can add, search, list, and delete contacts through a simple menu interface. The program is split into three modules: a **data module** that defines the contact record and handles file storage, an **I/O module** that handles menus, input, and display, and a **main module** that brings everything together.

23.1 Design First

Before writing any code, we must understand the problem and decide how to organise the solution. The guiding principle is **separation of concerns**: each module should have a single, clearly defined responsibility. Modules that try to do too many things become hard to understand and hard to change.

What Does the Program Need to Do?

The user should be able to:

1. **Add** a new contact (name, phone, email).
2. **List** all contacts in alphabetical order by name.
3. **Search** for a contact by name.
4. **Delete** a contact by name.
5. **Quit** — saving the contact list to a file automatically.

On startup, the program loads any previously saved contacts from the file. If the file does not yet exist, the program starts with an empty list.

Identifying the Modules

Three modules emerge naturally from this description:

`ABData` — the data module. It owns the contact records and the array that holds them. It knows nothing about menus or user interaction; it simply provides procedures for adding, finding, deleting, sorting, saving, and loading contacts. Any program that wanted to use a different interface — a graphical one, say — could import `ABData` unchanged.

`ABIO` — the I/O module. It handles all interaction with the user: displaying the menu, reading input, printing the contact list, and reporting errors. It imports `ABData` so it can display contacts, but it contains no file handling or data management logic of its own.

`AddressBook` — the main module. Its `BEGIN...END` block is the entry point of the program. It loads the data, runs the menu loop, and saves the data on exit. It is the glue between `ABData` and `ABIO`.

The Contact Record

Each contact has three string fields:

```
Contact = RECORD
  name  : ARRAY 30 OF CHAR;
  phone : ARRAY 20 OF CHAR;
  email : ARRAY 40 OF CHAR
END;
```

Contacts are stored in a fixed-size array of at most `MaxContacts` entries, with a separate integer `count` tracking how many are currently in use. This is the familiar pattern from Chapters 10 and 13.

Data Flow Between Modules

```
AddressBook (main)
    |
    |-- imports --> ABData   (data + file I/O)
    |-- imports --> ABIO     (menus + display)
                    |
                    |-- imports --> ABData (to display contacts)
```

`ABData` is at the bottom — it imports nothing from the other two modules. `ABIO` imports `ABData` so it can display contact data. `AddressBook` imports both and coordinates between them. This hierarchy means there are no circular dependencies.

23.2 The Data Module — ABData

`ABData` is responsible for all contact data and file operations. Its exported interface is:

- The `Contact` type and `ContactArr` array type, so `ABIO` can display them.
- The `contacts` array and `count` variable (read-only to clients).
- `Add` — add a new contact.
- `Delete` — delete a contact by index.
- `Find` — search for a contact by name, returning its index or -1.
- `Sort` — sort the array alphabetically by name.
- `Save` — save all contacts to a file.
- `Load` — load contacts from a file (if it exists).

```
MODULE ABData;
  IMPORT Files, Strings, Out;

CONST
  MaxContacts* = 50;
  NameLen      = 30;
  PhoneLen     = 20;
```

```
  EmailLen     = 40;
  FileName     = "contacts.dat";

TYPE
  NameStr*   = ARRAY NameLen  OF CHAR;
  PhoneStr*  = ARRAY PhoneLen OF CHAR;
  EmailStr*  = ARRAY EmailLen OF CHAR;

  Contact* = RECORD
    name*  : NameStr;
    phone* : PhoneStr;
    email* : EmailStr
  END;

  ContactArr* = ARRAY MaxContacts OF Contact;

VAR
  contacts* : ContactArr;
  count*    : INTEGER;

(*-------------------------------------------------------------*)

PROCEDURE Add*(name : NameStr; phone : PhoneStr; email : EmailStr);
BEGIN
  IF count < MaxContacts THEN
    contacts[count].name  := name;
    contacts[count].phone := phone;
    contacts[count].email := email;
    INC(count)
  ELSE
    Out.String("Address book is full.");
    Out.Ln
  END
END Add;

(*-------------------------------------------------------------*)

PROCEDURE Find*(name : NameStr) : INTEGER;
VAR
  i      : INTEGER;
  result : INTEGER;
BEGIN
  result := -1;
  i      := 0;
  WHILE (i < count) & (result = -1) DO
    IF contacts[i].name = name THEN
```

```
      result := i
    END;
    INC(i)
  END;
  RETURN result
END Find;

(*-------------------------------------------------------------*)

PROCEDURE Delete*(index : INTEGER);
VAR
  i : INTEGER;
BEGIN
  IF (index >= 0) & (index < count) THEN
    FOR i := index TO count - 2 DO
      contacts[i] := contacts[i + 1]
    END;
    DEC(count)
  END
END Delete;

(*-------------------------------------------------------------*)

PROCEDURE Sort*;
VAR
  i, j, minPos : INTEGER;
  temp         : Contact;
BEGIN
  FOR i := 0 TO count - 2 DO
    minPos := i;
    FOR j := i + 1 TO count - 1 DO
      IF contacts[j].name < contacts[minPos].name THEN
        minPos := j
      END
    END;
    IF minPos # i THEN
      temp             := contacts[i];
      contacts[i]      := contacts[minPos];
      contacts[minPos] := temp
    END
  END
END Sort;

(*-------------------------------------------------------------*)

PROCEDURE Save*;
```

```
VAR
  f : Files.File;
  r : Files.Rider;
  i : INTEGER;
BEGIN
  f := Files.New(FileName);
  Files.Set(r, f, 0);
  Files.WriteInt(r, count);
  FOR i := 0 TO count - 1 DO
    Files.WriteString(r, contacts[i].name);
    Files.WriteString(r, contacts[i].phone);
    Files.WriteString(r, contacts[i].email)
  END;
  Files.Register(f)
END Save;

(*------------------------------------------------------------*)

PROCEDURE Load*;
VAR
  f : Files.File;
  r : Files.Rider;
  i : INTEGER;
BEGIN
  f := Files.Old(FileName);
  IF f # NIL THEN
    Files.Set(r, f, 0);
    Files.ReadInt(r, count);
    IF count > MaxContacts THEN count := MaxContacts END;
    FOR i := 0 TO count - 1 DO
      Files.ReadString(r, contacts[i].name);
      Files.ReadString(r, contacts[i].phone);
      Files.ReadString(r, contacts[i].email)
    END
  END
END Load;

(*------------------------------------------------------------*)

BEGIN
  count := 0
END ABData.
```

A few design decisions are worth noting.

`count` is exported with `*` making it readable by clients, but since exported variables are read-only to clients in Oberon-07 only `ABData`'s own procedures can modify it. This protects the count from being

accidentally changed by other modules.

`Save` writes the count as an integer first, then the contact strings. `Load` reads the count first, then reads exactly that many contacts. This is simpler and more robust than looping until `eof` — the count tells us exactly how many records to expect.

`Delete` closes the gap left by the deleted contact by shifting all subsequent contacts one position to the left. This preserves the sorted order.

`Sort` uses the selection sort algorithm from Chapter 10, comparing the `name` fields of `Contact` records directly.

23.3 The I/O Module — ABIO

`ABIO` handles all interaction with the user. It imports `ABData` to access the contact array and count, but contains no data management or file handling logic of its own.

```
MODULE ABIO;
  IMPORT In, Out, ABData;

(*------------------------------------------------------------*)

PROCEDURE ReadLine(prompt : ARRAY OF CHAR;
                   VAR s  : ARRAY OF CHAR);
BEGIN
  Out.String(prompt);
  In.Line(s)
END ReadLine;

(*------------------------------------------------------------*)

PROCEDURE PrintContact*(i : INTEGER);
BEGIN
  Out.String("  Name:  "); Out.String(ABData.contacts[i].name);  Out.Ln;
  Out.String("  Phone: "); Out.String(ABData.contacts[i].phone); Out.Ln;
  Out.String("  Email: "); Out.String(ABData.contacts[i].email); Out.Ln
END PrintContact;

(*------------------------------------------------------------*)

PROCEDURE ListAll*;
VAR
  i : INTEGER;
BEGIN
  IF ABData.count = 0 THEN
    Out.String("No contacts.");
    Out.Ln
  ELSE
    Out.Ln;
```

```
    FOR i := 0 TO ABData.count - 1 DO
      Out.Int(i + 1, 3);
      Out.String(". ");
      Out.String(ABData.contacts[i].name);
      Out.Ln
    END
  END
END ListAll;

(*------------------------------------------------------------*)

PROCEDURE ShowContact*(i : INTEGER);
BEGIN
  Out.Ln;
  PrintContact(i);
  Out.Ln
END ShowContact;

(*------------------------------------------------------------*)

PROCEDURE AddContact*;
VAR
  name  : ABData.NameStr;
  phone : ABData.PhoneStr;
  email : ABData.EmailStr;
BEGIN
  Out.Ln;
  ReadLine("Name:  ", name);
  IF name[0] # 0X THEN
    ReadLine("Phone: ", phone);
    ReadLine("Email: ", email);
    ABData.Add(name, phone, email);
    ABData.Sort;
    Out.String("Contact added.");
    Out.Ln
  ELSE
    Out.String("Cancelled.");
    Out.Ln
  END
END AddContact;

(*------------------------------------------------------------*)

PROCEDURE SearchContact*;
VAR
  name  : ABData.NameStr;
```

```
  index : INTEGER;
BEGIN
  Out.Ln;
  ReadLine("Search name: ", name);
  index := ABData.Find(name);
  IF index >= 0 THEN
    ShowContact(index)
  ELSE
    Out.String("'");
    Out.String(name);
    Out.String("' not found.");
    Out.Ln
  END
END SearchContact;

(*-------------------------------------------------------------*)

PROCEDURE DeleteContact*;
VAR
  name  : ABData.NameStr;
  index : INTEGER;
  conf  : ARRAY 5 OF CHAR;
BEGIN
  Out.Ln;
  ReadLine("Delete name: ", name);
  index := ABData.Find(name);
  IF index >= 0 THEN
    ShowContact(index);
    ReadLine("Delete this contact? (y/n): ", conf);
    IF (conf[0] = "y") OR (conf[0] = "Y") THEN
      ABData.Delete(index);
      Out.String("Contact deleted.");
      Out.Ln
    ELSE
      Out.String("Cancelled.");
      Out.Ln
    END
  ELSE
    Out.String("'");
    Out.String(name);
    Out.String("' not found.");
    Out.Ln
  END
END DeleteContact;

(*-------------------------------------------------------------*)
```

```
PROCEDURE Menu*;
BEGIN
  Out.Ln;
  Out.String("1. Add contact");    Out.Ln;
  Out.String("2. List all");       Out.Ln;
  Out.String("3. Search");         Out.Ln;
  Out.String("4. Delete");         Out.Ln;
  Out.String("5. Quit");           Out.Ln;
  Out.String("Choice: ")
END Menu;

END ABIO.
```

`ReadLine` is a private helper — not exported — that prints a prompt and reads a line of input. All the exported procedures use it rather than calling `Out.String` and `In.Line` directly, which keeps the input style consistent throughout.

`AddContact` sorts the list after adding each new contact, so the list is always in alphabetical order. An alternative would be to sort only when displaying, but sorting on insertion means `Find` could be extended to use binary search in future if the list becomes large.

`DeleteContact` asks the user to confirm before deleting, showing the contact's details first. This guards against accidentally deleting the wrong contact.

23.4 The Main Module — AddressBook

The main module's job is simple: load the data, run the menu loop, and save on exit. All the real work is delegated to `ABData` and `ABIO`.

```
MODULE AddressBook;
  IMPORT In, Out, ABData, ABIO;
VAR
  choice : INTEGER;
  done   : BOOLEAN;
  flush  : ARRAY 10 OF CHAR;

BEGIN
  ABData.Load;
  Out.String("Address Book (");
  Out.Int(ABData.count, 0);
  Out.String(" contact(s) loaded)");
  Out.Ln;
  done := FALSE;
  WHILE ~done DO
    ABIO.Menu;
    In.Int(choice);
```

```
    In.Line(flush);    (* consume leftover newline *)
    IF ~In.Done THEN
      done := TRUE
    ELSIF choice = 1 THEN
      ABIO.AddContact
    ELSIF choice = 2 THEN
      ABIO.ListAll
    ELSIF choice = 3 THEN
      ABIO.SearchContact
    ELSIF choice = 4 THEN
      ABIO.DeleteContact
    ELSIF choice = 5 THEN
      done := TRUE
    ELSE
      Out.String("Please enter 1-5.");
      Out.Ln
    END
  END;
  ABData.Save;
  Out.String("Contacts saved. Goodbye.");
  Out.Ln
END AddressBook.
```

The main block is deliberately short; a reader can understand the entire structure of the program just by reading this module. The details are in `ABData` and `ABIO`, where they can be studied one piece at a time.

Note that `In.Line(flush)` is required to avoid the newline-flushing problem from Chapter 19. Without it, after `In.Int(choice)` reads the menu choice, the newline from the `Enter` keypress would remain in the input buffer. Then when `ABIO.AddContact` calls `ReadLine` which uses `In.Line`, it would immediately read the leftover newline and get an empty string, thus triggering the "Cancelled" message.

23.5 A Complete Walkthrough

Let us trace through a typical session to confirm the program behaves correctly.

Startup

The program calls `ABData.Load`. If `contacts.dat` does not exist, `Files.Old` returns `NIL` and `Load` returns immediately with `count = 0` (set in `ABData`'s initialisation block). If the file exists, the contacts are read in and `count` is set to the number of records found.

Adding a Contact

The user selects option 1. `ABIO.AddContact` is called:

```
1. Add contact
2. List all
3. Search
4. Delete
5. Quit
Choice: 1

Name:  Alice Smith
Phone: 07700900123
Email: alice@example.com
Contact added.
```

`AddContact` reads the three fields using `ReadLine`, calls `ABData.Add` to store the contact, then calls `ABData.Sort` to keep the list alphabetical. If the user presses Enter on a blank name, the entry is cancelled — `name[0] = 0X` detects the empty string.

Listing Contacts

After adding a few contacts, option 2 shows:

```
Choice: 2

  1. Alice Smith
  2. Bob Jones
  3. Carol White
```

`ABIO.ListAll` prints each contact's name with a sequence number. The list is already sorted because `AddContact` sorted it on insertion.

Searching

```
Choice: 3

Search name: Bob Jones

  Name:  Bob Jones
  Phone: 07700900456
  Email: bob@example.com
```

`ABIO.SearchContact` calls `ABData.Find`, which does a linear scan of the name fields. If found, `ABIO.ShowContact` prints the full details.

Deleting

```
Choice: 4

Delete name: Bob Jones

  Name:  Bob Jones
  Phone: 07700900456
  Email: bob@example.com

Delete this contact? (y/n): y
Contact deleted.
```

`ABIO.DeleteContact` finds the contact, displays it, asks for confirmation, and calls `ABData.Delete` if confirmed. `Delete` shifts subsequent contacts left to close the gap.

Quitting and Saving

```
Choice: 5
Contacts saved. Goodbye.
```

The main loop detects `choice = 5`, sets `done := TRUE`, exits the loop, and calls `ABData.Save`. The contacts are written to `contacts.dat` and will be loaded automatically next time the program runs.

23.6 Compiling and Running

All four files — `ABData.obn`, `ABIO.obn`, `AddressBook.obn` — must be in the same directory. Compile by naming the main module:

```
obnc AddressBook.obn
./AddressBook
```

OBNC detects the dependencies automatically and compiles all three modules in the correct order.

23.7 Possible Extensions

The address book as written is functional but deliberately simple. Here are some natural extensions to consider as exercises:

Case-insensitive search. The current `Find` procedure uses exact string matching — `"Alice"` and `"alice"` are treated as different names. Using `Strings.Cap` on both the search term and the stored name before comparing would make search case-insensitive.

Partial name search. `Strings.Pos` could be used to find contacts whose name *contains* the search string, rather than exactly matching it. This is more forgiving for users who cannot remember the exact spelling.

Editing a contact. Add an option to modify an existing contact's phone or email. This would require a new `Update` procedure in `ABData` and a corresponding `EditContact` procedure in `ABIO`.

Duplicate detection. `ABData.Add` currently allows two contacts with the same name. Calling `Find` before `Add` and warning the user if the name already exists would be a simple improvement.

Binary search. Since the list is always kept sorted, `Find` could use binary search (Chapter 15) instead of a linear scan, making it significantly faster for large lists.

Summary

Building a larger program requires sustained design effort before coding begins. The key decisions for the address book were:

- **Separation of concerns**: data management in `ABData`, user interaction in `ABIO`, coordination in `AddressBook`. Each module has one clear responsibility.
- **Design the interface before the implementation**: knowing what each module exports — and what it does not — before writing any procedure bodies keeps the design clean and prevents unwanted dependencies.
- **The main module should be short**: if the main `BEGIN...END` block is long and complex, that is a sign that logic has not been moved into appropriate procedures and modules.
- **Save on exit, load on startup**: the simplest possible persistence strategy — no complex transaction management, no partial saves.

These principles apply to programs of any size. The address book is small enough to hold in your head all at once, but the same approach — modular decomposition, clean interfaces, short main block — scales to programs of thousands of lines.

Exercises

23.1 Add case-insensitive search to `ABData.Find` using `Strings.Cap`. Test it by searching for `"alice"` when the stored name is `"Alice Smith"`.

23.2 Add an `Edit` procedure to `ABData` that takes an index and new values for the phone and email fields, and a corresponding `EditContact` procedure to `ABIO` that finds a contact by name and allows the user to update its details. Add option 6 to the menu.

23.3 Add duplicate detection to `ABIO.AddContact`: before calling `ABData.Add`, call `ABData.Find` and warn the user if a contact with that name already exists, asking whether to proceed.

23.4 Replace the linear search in `ABData.Find` with binary search (from Chapter 15). Since the list is always kept sorted, binary search is correct here. Verify that search still works correctly after adding and deleting contacts.

23.5 Add a `Count` display to the menu header that shows how many contacts are currently stored, so the user always knows the list size at a glance.

23.6 Add a partial-match search option to `ABIO`: prompt the user for a fragment of a name, then list all contacts whose name *contains* that fragment using `Strings.Pos`. This should be a separate menu option from the exact-match search.

23.7 Extend the `Contact` record in `ABData` to include a fourth field: `address : ARRAY 60 OF CHAR`. Update `Add`, `Save`, `Load`, `PrintContact`, and `AddContact` accordingly. Note that adding the field will make any existing `contacts.dat` file unreadable — you will need to delete it and start fresh.

Appendix A — Reserved Words & Symbols

A.1 Reserved Words

The following words are reserved in Oberon-07. They consist exclusively of capital letters and cannot be used as identifiers:

```
ARRAY     BEGIN     BY        CASE      CONST
DIV       DO        ELSE      ELSIF     END
FALSE     FOR       IF        IMPORT    IN
IS        MOD       MODULE    NIL       OF
OR        POINTER   PROCEDURE RECORD    REPEAT
RETURN    THEN      TO        TRUE      TYPE
UNTIL     VAR       WHILE
```

A.2 Operators and Delimiters

The following special characters and character pairs are operators or delimiters:

Symbol	Meaning
`+`	Addition; set union
`-`	Subtraction; set difference; unary minus; set complement
`*`	Multiplication; set intersection
`/`	Real division; symmetric set difference
`~`	Logical NOT
`&`	Logical AND
`=`	Equal
`#`	Not equal
`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal
`IN`	Set membership
`IS`	Type test
`OR`	Logical OR
`DIV`	Integer division
`MOD`	Integer remainder
`:=`	Assignment

Symbol	Meaning
`^`	Pointer dereference
`.`	Field selector; module qualifier; end of module
`,`	Separator in lists
`;`	Statement separator
`:`	Type separator
`|`	Case separator
`()`	Parameter list; subexpression
`[]`	Array index
`{}`	Set literal
`..`	Range in set literal or CASE
`(**)`	Comment delimiters
`*`	Export marker (in declarations)

A.3 Operator Precedence

Operators are grouped into four precedence levels. Higher in the table means higher precedence (binds more tightly). Operators at the same level associate left to right.

Precedence	Operators
Highest	~ (unary NOT)
2nd	`* / DIV MOD &`
3rd	`+ - OR`
Lowest	`= # < <= > >= IN IS`

A.4 Lexical Rules

Identifiers are sequences of letters and digits beginning with a letter. Upper and lower case letters are distinct: `count`, `Count`, and `COUNT` are three different identifiers.

Integer literals are sequences of decimal digits, optionally followed by `H` to indicate hexadecimal: `42`, `255`, `0FFH`.

Real literals contain a decimal point and optionally a scale factor: `3.14`, `1.5E10`, `2.0E-3`.

Character literals are either a single character in double quotes (`"A"`) or a two-digit hexadecimal ordinal followed by `X` (`41X` = `"A"`).

String literals are sequences of characters enclosed in double quotes: `"Hello, World!"`. A string cannot contain the double-quote character itself.

Comments are enclosed in `(*` and `*)` and may be nested. They can appear between any two symbols and have no effect on the program.

Appendix B — Predefined Identifiers & Procedures

Predefined identifiers are built into the language and do not need to be declared or imported. They may be redeclared by the programmer, but this is strongly discouraged.

B.1 Predefined Types

Identifier	Description
`BOOLEAN`	The truth values `TRUE` and `FALSE`
`CHAR`	Characters of the ASCII character set
`INTEGER`	Integer numbers (64-bit signed on a 64-bit system)
`REAL`	Real (floating-point) numbers (64-bit IEEE 754)
`BYTE`	Integers in the range 0 to 255; compatible with `INTEGER`
`SET`	Sets of integers from 0 to 63 (on a 64-bit system)

B.2 Predefined Constants

Identifier	Value	Description
`TRUE`	`TRUE`	Boolean true value
`FALSE`	`FALSE`	Boolean false value
`NIL`	—	Null pointer value

B.3 Predefined Function Procedures

Function procedures return a value and are used in expressions.

Numeric Functions

Call	Argument type	Result type	Description
`ABS(x)`	Numeric	Same as `x`	Absolute value of `x`
`ODD(x)`	`INTEGER`	`BOOLEAN`	`TRUE` if `x` is odd
`LEN(a)`	Any array	`INTEGER`	Number of elements in array `a`

Shift and Rotation Functions

Call	Argument types	Result type	Description
`LSL(x, n)`	`INTEGER`, `INTEGER`	`INTEGER`	Logical shift left: `x * 2^n`

Call	Argument types	Result type	Description
`ASR(x, n)`	`INTEGER`, `INTEGER`	`INTEGER`	Arithmetic shift right: `x DIV 2^n`
`ROR(x, n)`	`INTEGER`, `INTEGER`	`INTEGER`	Rotate `x` right by `n` bits

Type Conversion Functions

Call	Argument type	Result type	Description
`FLOOR(x)`	`REAL`	`INTEGER`	Largest integer not greater than `x`
`FLT(n)`	`INTEGER`	`REAL`	Convert integer to real
`ORD(x)`	`CHAR`, `BOOLEAN`, or `SET`	`INTEGER`	Ordinal value of `x`
`CHR(n)`	`INTEGER`	`CHAR`	Character with ordinal value `n`

Note: `FLOOR(1.5) = 1` and `FLOOR(-1.5) = -2`.

B.4 Predefined Proper Procedures

Proper procedures perform an action but do not return a value.

Integer Increment and Decrement

Call	Types	Effect
`INC(v)`	`v : INTEGER`	`v := v + 1`
`INC(v, n)`	`v, n : INTEGER`	`v := v + n`
`DEC(v)`	`v : INTEGER`	`v := v - 1`
`DEC(v, n)`	`v, n : INTEGER`	`v := v - n`

Set Operations

Call	Types	Effect
`INCL(v, x)`	`v : SET; x : INTEGER`	`v := v + {x}`
`EXCL(v, x)`	`v : SET; x : INTEGER`	`v := v - {x}`

Dynamic Memory

Call	Types	Effect
`NEW(v)`	`v : pointer`	Allocates a new record; stores address in `v`. If allocation fails, `v` is set to `NIL`.

Assertions

Call	Types	Effect
`ASSERT(b)`	`b : BOOLEAN`	Traps with exception code 8 if `b` is `FALSE`
`ASSERT(b, n)`	`b : BOOLEAN; n : INTEGER`	Traps with exception code `n` if `b` is `FALSE`

Real Number Packing

These procedures are low-level operations on the internal representation of real numbers. They are rarely needed in ordinary programs.

Call	Types	Effect
`PACK(x, n)`	`x : REAL; n : INTEGER`	Equivalent to `x := x * 2^n`
`UNPK(x, n)`	`x : REAL; n : INTEGER`	Sets `x` to the normalised mantissa ($1.0 \le x < 2.0$) and `n` to the exponent

---

Appendix C — OBNC Library Reference

This appendix lists the procedures and variables exported by each standard OBNC library module. For full descriptions, consult the individual chapters where each module is introduced.

In — Standard Input

Introduced in Chapter 4.

```
IMPORT In;
```

Identifier	Description
`In.Done`	`BOOLEAN` — `TRUE` if the last read operation succeeded
`In.Open`	Included for compatibility; does nothing in practice
`In.Char(ch)`	Reads one character (does not skip whitespace)
`In.Int(i)`	Reads an integer (skips leading whitespace)
`In.Real(x)`	Reads a real number (skips leading whitespace)
`In.String(s)`	Reads a quoted string
`In.Name(s)`	Reads a sequence of non-whitespace characters
`In.Line(s)`	Reads to end of line (does not skip leading whitespace)

Out — Standard Output

Introduced in Chapter 4.

```
IMPORT Out;
```

Identifier	Description
`Out.Open`	Included for compatibility; does nothing
`Out.Char(ch)`	Writes character `ch`
`Out.String(s)`	Writes string `s` (without the null terminator)
`Out.Int(i, n)`	Writes integer `i` right-justified in field width `n`
`Out.Real(x, n)`	Writes real `x` in scientific notation, field width `n`
`Out.Hex(i)`	Writes integer `i` as zero-padded hexadecimal
`Out.Ln`	Writes a newline

Strings — String Operations

Introduced in Chapter 9.

```
IMPORT Strings;
```

All procedures operate on null-terminated character arrays. Positions are zero-based.

Identifier	Description
`Strings.Length(s)`	Returns the length of `s` (excluding `0X`)
`Strings.Append(extra, dest)`	Appends `extra` to `dest`
`Strings.Insert(src, pos, dest)`	Inserts `src` into `dest` at position `pos`
`Strings.Delete(s, pos, n)`	Deletes `n` characters from `s` at `pos`
`Strings.Replace(src, pos, dest)`	Replaces characters in `dest` at `pos` with `src`
`Strings.Extract(src, pos, n, dest)`	Copies `n` characters from `src` at `pos` to `dest`
`Strings.Pos(pattern, s, pos)`	Returns position of first match of `pattern` in `s` from `pos`; `-1` if not found
`Strings.Cap(s)`	Converts all lowercase letters in `s` to uppercase

Files — File Operations

Introduced in Chapter 19.

```
IMPORT Files;
```

Identifier	Description
`Files.File`	Type: handle to an open file
`Files.Rider`	Type: read/write position marker; fields `eof`, `res`
`Files.Old(name)`	Opens existing file; returns `NIL` if not found
`Files.New(name)`	Creates new file in memory (not yet on disk)
`Files.Register(f)`	Writes file `f` to disk
`Files.Close(f)`	Flushes buffers of file `f`
`Files.Delete(name, res)`	Removes file from directory; `res = 0` on success
`Files.Rename(old, new, res)`	Renames file; `res = 0` on success
`Files.Length(f)`	Returns file length in bytes
`Files.Set(r, f, pos)`	Positions rider `r` at byte `pos` in file `f`
`Files.Pos(r)`	Returns current position of rider `r`

Identifier	Description
`Files.GetError(msg)`	Returns error message for last failed operation
`Files.Read(r, x)`	Reads one byte
`Files.ReadInt(r, i)`	Reads one integer
`Files.ReadReal(r, x)`	Reads one real number
`Files.ReadString(r, s)`	Reads a null-terminated string
`Files.ReadBool(r, b)`	Reads one Boolean value
`Files.ReadSet(r, s)`	Reads one set value
`Files.ReadNum(r, i)`	Reads a compactly encoded integer
`Files.ReadBytes(r, buf, n)`	Reads `n` bytes into `buf`
`Files.Write(r, x)`	Writes one byte
`Files.WriteInt(r, i)`	Writes one integer
`Files.WriteReal(r, x)`	Writes one real number
`Files.WriteString(r, s)`	Writes a null-terminated string
`Files.WriteBool(r, b)`	Writes one Boolean value
`Files.WriteSet(r, s)`	Writes one set value
`Files.WriteNum(r, i)`	Writes a compactly encoded integer
`Files.WriteBytes(r, buf, n)`	Writes `n` bytes from `buf`

Math — Mathematical Functions

Introduced in Chapter 18.

```
IMPORT Math;
```

Identifier	Description
`Math.pi`	π ≈ 3.14159265358979
`Math.e`	e ≈ 2.71828182845905
`Math.sqrt(x)`	Square root of `x` (`x ≥ 0`)
`Math.power(base, exp)`	`base` raised to `exp`
`Math.round(x)`	`x` rounded to nearest integer (returned as `REAL`)
`Math.exp(x)`	e raised to `x`
`Math.ln(x)`	Natural logarithm of `x` (`x > 0`)
`Math.log(x, base)`	Logarithm of `x` with given `base`
`Math.sin(x)`	Sine of `x` (radians)

Identifier	Description
`Math.cos(x)`	Cosine of `x` (radians)
`Math.tan(x)`	Tangent of `x` (radians)
`Math.arcsin(x)`	Inverse sine, result in radians (`-1 ≤ x ≤ 1`)
`Math.arccos(x)`	Inverse cosine, result in radians (`-1 ≤ x ≤ 1`)
`Math.arctan(x)`	Inverse tangent, result in radians
`Math.arctan2(y, x)`	Angle of point `(x, y)`, result in radians
`Math.sinh(x)`	Hyperbolic sine
`Math.cosh(x)`	Hyperbolic cosine
`Math.tanh(x)`	Hyperbolic tangent
`Math.arcsinh(x)`	Inverse hyperbolic sine
`Math.arccosh(x)`	Inverse hyperbolic cosine
`Math.arctanh(x)`	Inverse hyperbolic tangent

---

extConvert — Number/String Conversions

Introduced in Chapter 17.

```
IMPORT extConvert;
```

Identifier	Description
`extConvert.IntToString(i, s, done)`	Converts integer `i` to decimal string `s`
`extConvert.RealToString(x, s, done)`	Converts real `x` to string `s`
`extConvert.StringToInt(s, i, done)`	Parses `s` as an integer into `i`
`extConvert.StringToReal(s, x, done)`	Parses `s` as a real number into `x`

All four procedures set `done` to `TRUE` on success and `FALSE` on failure.

---

extArgs — Command-Line Arguments

Introduced in Chapter 17.

```
IMPORT extArgs;
```

Identifier	Description
`extArgs.count`	Number of command-line arguments (read-only)

Identifier	Description
`extArgs.Get(n, arg, res)`	Retrieves argument `n` (0-based) into `arg`; `res` is the number of truncated characters

---

extErr — Standard Error Output

Introduced in Chapter 20.

```
IMPORT extErr;
```

Mirrors `Out` but writes to the standard error stream.

Identifier	Description
`extErr.Char(ch)`	Writes character `ch` to standard error
`extErr.String(s)`	Writes string `s` to standard error
`extErr.Int(i, n)`	Writes integer `i`, field width `n`, to standard error
`extErr.Real(x, n)`	Writes real `x` in scientific notation to standard error
`extErr.Hex(i)`	Writes integer `i` as hexadecimal to standard error
`extErr.Ln`	Writes a newline to standard error

---

extTrap — Custom Trap Handlers

Introduced in Chapter 20.

```
IMPORT extTrap;
```

Identifier	Description
`extTrap.Handler`	Type: `PROCEDURE(exception : INTEGER; file : ARRAY OF CHAR; line : INTEGER)`
`extTrap.handle`	The currently installed handler (read-only)
`extTrap.SetHandler(h)`	Installs `h` as the trap handler (`h` must not be `NIL`)

Exception codes:

Code	Meaning
1	Destination array too short for assignment
2	Array index out of bounds

Code	Meaning
3	NIL pointer dereference
4	NIL procedure variable call
5	Source in assignment is not an extension of target
6	Type guard failure
7	Unmatched expression in CASE statement
8	Assertion failure

---

extPipes — Pipe Communication

Introduced in Chapter 21.

```
IMPORT extPipes;
```

Identifier	Description
`extPipes.Stream`	Type: pointer to record with field `eof : BOOLEAN`
`extPipes.OpenRead(command, stream)`	Runs `command`; connects `stream` to its stdout
`extPipes.OpenWrite(command, stream)`	Runs `command`; connects `stream` to its stdin
`extPipes.Read(stream, ch)`	Reads one character from `stream`
`extPipes.Write(ch, stream)`	Writes one character to `stream`
`extPipes.Close(stream, exitCode)`	Closes `stream`; returns command's exit code

`stream` is set to `NIL` if `OpenRead` or `OpenWrite` fails. `stream.eof` becomes `TRUE` at end of output or on write failure.

XYplane — Pixel Graphics

Introduced in Chapter 21.

```
IMPORT XYplane;
```

Identifier	Description
`XYplane.draw`	Constant `1`: draw mode for `Dot`
`XYplane.erase`	Constant `0`: erase mode for `Dot`
`XYplane.W`	Width of the drawing plane in pixels (read-only)
`XYplane.H`	Height of the drawing plane in pixels (read-only)

Identifier	Description
`XYplane.SetSize(w, h)`	Sets window size before `Open`
`XYplane.Open`	Opens the graphics window
`XYplane.Clear`	Erases all pixels
`XYplane.Dot(x, y, mode)`	Draws or erases pixel at `(x, y)` (origin: lower-left)
`XYplane.IsDot(x, y)`	Returns `TRUE` if pixel at `(x, y)` is drawn
`XYplane.Key()`	Repaints window; returns any key pressed, else `0X`
`XYplane.UseColor(color)`	Sets drawing colour as 24-bit RGB integer
`XYplane.Color(x, y)`	Returns colour of pixel at `(x, y)`

Appendix D — ASCII Character Table

ASCII (American Standard Code for Information Interchange) defines 128 characters, numbered 0 to 127. Ordinals 0 to 31 and 127 are non-printable control characters; ordinals 32 to 126 are printable characters. In Oberon-07, `ORD(ch)` returns the ASCII ordinal of character `ch`, and `CHR(n)` returns the character with ordinal `n`.

Printable Characters (32–126)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	(space)	56	38	8	80	50	P	104	68	h
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	$	60	3C	<	84	54	T	108	6C	l
37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	'	63	3F	?	87	57	W	111	6F	o
40	28	(	64	40	@	88	58	X	112	70	p
41	29	)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[	115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D	]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	`	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	|
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g	127	7F	DEL

Control Characters (0–31)

These characters are non-printable. The most commonly encountered in Oberon-07 programs are `NUL` (0, the null terminator `0X`), `HT` (9, the tab character), `LF` (10, the newline `0AX`), and `CR` (13, the carriage return `0DX`).

Dec	Hex	Name	Meaning	Dec	Hex	Name	Meaning
0	00	NUL	Null (string terminator `0X`)	16	10	DLE	Data Link Escape
1	01	SOH	Start of Heading	17	11	DC1	Device Control 1
2	02	STX	Start of Text	18	12	DC2	Device Control 2
3	03	ETX	End of Text	19	13	DC3	Device Control 3
4	04	EOT	End of Transmission	20	14	DC4	Device Control 4
5	05	ENQ	Enquiry	21	15	NAK	Negative Acknowledge
6	06	ACK	Acknowledge	22	16	SYN	Synchronous Idle
7	07	BEL	Bell	23	17	ETB	End of Trans. Block
8	08	BS	Backspace	24	18	CAN	Cancel
9	09	HT	Horizontal Tab	25	19	EM	End of Medium
10	0A	LF	Line Feed (newline `0AX`)	26	1A	SUB	Substitute
11	0B	VT	Vertical Tab	27	1B	ESC	Escape
12	0C	FF	Form Feed	28	1C	FS	File Separator
13	0D	CR	Carriage Return `0DX`	29	1D	GS	Group Separator
14	0E	SO	Shift Out	30	1E	RS	Record Separator
15	0F	SI	Shift In	31	1F	US	Unit Separator

Useful Ranges for Oberon-07

The following ranges are frequently used in character comparisons:

Range	Ordinals	Test
Uppercase A–Z	65–90	`(ch >= "A") & (ch <= "Z")`
Lowercase a–z	97–122	`(ch >= "a") & (ch <= "z")`
Digits 0–9	48–57	`(ch >= "0") & (ch <= "9")`
Printable	32–126	`(ch >= " ") & (ch <= "~")`

The difference between the ordinal of a lowercase letter and its uppercase equivalent is always 32: `ORD("a") - ORD("A") = 32`.